Representing short sequences in the context of a model organism genome

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Christopher T. Lewis

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

In the post-genomics era, the sheer volume of data is overwhelming without appropriate tools for data integration and analysis. Studying genomic sequences in the context of other related genomic sequences, i.e. comparative genomics, is a powerful technique enabling the identification of functionally interesting sequence regions based on the principal that similar sequences tend to be either homologous or provide similar functionality.

Costs associated with full genome sequencing make it infeasible to sequence every genome of interest. Consequently, simple, smaller genomes are used as model organisms for more complex organisms, for instance, Mouse/Human. An annotated model organism provides a source of annotation for transcribed sequences and other gene regions of the more complex organism based on sequence homology. For example, the gene annotations from the model organism aid interpretation of expression studies in more complex organisms.

To assist with comparative genomics research in the *Arabidopsis*/*Brassica* (Thale-cress/Canola) model-crop pair, a web-based, graphical genome browser (BioViz) was developed to display short *Brassica* genomic sequences in the context of the *Arabidopsis* model organism genome. This involved the development of graphical representations to integrate data from multiple sources and tools, and a novel user interface to provide the user with a more interactive web-based browsing experience. While BioViz was developed for the *Arabidopsis*/*Brassica* comparative genomics context, it could be applied to comparative browsing relative to other reference genomes.

BioViz proved to be an valuable research support tool for *Brassica* / *Arabidopsis* comparative genomics. It provided convenient access to the underlying *Arabidopsis* annotation, allowed the user to view specific EST sequences in the context of the *Arabidopsis* genome and other related EST sequences. In addition, the limits to which the project pushed the SVG specification proved influential in the SVG community. The work done for BioViz inspired the definition of an open-source project to define standards for SVG based web applications and a standard framework for SVG based widget sets.

Acknowledgements

I would like to thank the following people for their assistance in this process:

- My supervisors, Drs. Anthony Kusalik and Isobel Parkin, for their input and guidance over the years.
- Dr. Steve Karcz for his input in the initial development of BioViz.
- Dustin Cram for his assistance with the BioViz server-side rewrite.
- The members of my committee, Drs. Mark Keil, Ian McQuillan, Joseph Angel, and Pat Covello for their feedback and support.
- My external examiner, Dr. Paul Lu, for his helpful suggestions at the defense.

To Tracy - thanks for the support, patience and occasional kick in the ass.

Contents

List of Tables

List of Figures

List of Abbreviations

AAFC	Agriculture and Agri-food Canada
AJAX	Asynchronous JavaScript and XML
BAC	Bacterial Artificial Chromosome
BLAST	Basic Local Alignment Search Tool
CGUI	Custom GUI
CSS	Cascading Style Sheets
DBA	Database Administrator
DOM	Document Object Model
DTD	Document Type Definition
EST	Expressed Sequence Tag
GMOD	Generic Model Organism Database
GBrowse	GMOD Generic Genome Browser
GO	Gene Ontology
GUI	Graphical User Interface
HSP	High Scoring Pair
JCVI	J. Craig Venter Institute
MSA	Multiple Sequence Alignment
NJ	Neighbor Joining
NW	Needleman-Wunsch (exact global sequence alignment algorithm)
OO	Object-Oriented
RC	Row-Column
SAGE	Serial Analysis of Gene Expression
SMIL	Synchronized Multimedia Integration Language
SPARK	SVG Programmers' Application Resource Kit
SVG	Scalable Vector Graphics
SVGZ	A compressed SVG image
SW	Smith-Waterman (exact local sequence alignment algorithm)
UI	User Interface
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

Chapter 1

Introduction

1.1 Background

Comparative genomics relies on the ability to identify regions of local similarity in the genomes of different species or strains of a species. This allows the identification of putative functional elements in the genome, for example gene coding and regulatory regions, based on the following two principals: first, that functional elements tend to evolve at slower rates than non-functional elements due to selective pressure on the organism to maintain the function, and second that similar sequences have similar functions. The latter principal enables transitive sequence annotation, i.e. assigning annotations to unknown sequences based on similarity to an existing annotated sequence. This is extremely helpful as an aid in assigning putative function to unknown sequences in other organisms.

However, while helpful in many cases, transitive annotation is also dangerous because it can result in the propagation of mis-annotations. The potential result is a cycle where the increased number of sequences sharing the annotation increases user confidence in the annotation, which results in further propagation of the error. Even if the mis-annotation is eventually corrected, the damage has already been done because numerous other sequences have now been incorrectly annotated. This could be mitigated by ensuring that all sequences contained a list of evidence for the annotation, but would still rely on the user to carefully review all entries. Consequently, tools to assist researchers in understanding the relationship between similar sequences are of fundamental importance in that they contribute to proper sequence identification and annotation.

When discussing sequence similarity, sequences are either globally similar (the sequences are similar over their full length), locally similar (the sequences have regions that are similar to each other), or lack similarity. For example, two or more genome sequences may contain regions of local similarity corresponding to homologous genes and other functional elements. However, the full genome sequences are likely to lack global similarity as a result of genome rearrangement. Even in regions with co-linear (i.e. non-rearranged) gene sequences, the inter-genic regions (excepting functional but non-coding regions) are relatively free to evolve, causing a loss of global similarity over time. Thus, the more closely related the sequences the greater the chance that they will

exhibit global similarity and the more conserved will be the co-linear regions. In the same way, short sequences such as coding sequences may or may not be globally similar depending on the evolutionary distance between sequences and the degree to which internal elements (e.g. domains) have undergone rearrangement.

Tools for identifying similar sequences are likely to perform a type of local alignment search. BLAST [1] (Basic Local Alignment and Search Tool) is a well known and commonly utilized tool for identifying sequence similarities. The program takes one or more "query" sequences and performs a seeded search against a pre-indexed database of "subject" sequences. The output is a report containing a set of local alignments for all query-subject pairs with similarities scoring above a predefined threshold. Each match, referred to as a "hit", consists of one or more high scoring pairs (HSPs) (or regions of local similarity). The pairwise alignments produced by BLAST are typical of the output from pairwise alignment programs, and they enable a fine-grained, position-by-position comparison of the sequences. However, in many cases a group of related sequences must be studied to get a sense of the relationship among sequences. This task is usually accomplished by performing a multiple sequence alignment (MSA).

MSA is a fundamental step in many biological applications, for example the design of specific oligonucleotide sequences for use in DNA microarrays, the identification of single nucleotide polymorphisms (SNPs) that distinguish closely related genes, the identification of protein family domains and motifs, and studies of evolutionary relatedness (which includes phylogenetic analysis and comparative genomics). Consequently, it is a well-studied computational problem, which has been shown to be NP-complete using the sum-of-pairs (SP) objective function [5, 60, 25] when the number of sequences is allowed to vary. The standard approach to overcoming the NP-complete nature of the problem is to perform a progressive alignment of the set of sequences. This is a type of "greedy" algorithm [10] that aligns sequences one pair at a time, typically the most closely related pair at each step as determined from a pre-computed guide tree. The intermediate and final alignments are generally scored using the SP objective function, and the final alignment is presented as a row-column (RC) alignment matrix. The well-known CLUSTAL suite of programs [55] popularized this approach, and the alignments produced by CLUSTALW are often used as the baseline for evaluating new MSA algorithms.

1.2 Problem

This thesis work focused on representing short *Brassica* sequences such as EST sequences in the context of the *Arabidopsis* genome to assist researchers in properly identifying the *Brassica* sequences. Standard bioinformatics tools such as BLAST can be used to identify regions in the *Arabidopsis* genome with similarity to a given *Brassica* EST; however, in many cases the information contained

in the report is insufficient for a proper understanding of the relationship. For instance, the hit between an EST and the genome sequence may contain any number of HSPs. In the simplest case, the EST will match a single locus and the hit will consist of a small number of HSPs corresponding to the alignment of the EST with the gene's exons. In this case, the report might contain enough information to elucidate the sequence relationship. However, depending on the relative length of the exons and the degree of conservation between the sequences, the HSPs may or may not occur in a linear order, which makes the relationship more difficult to interpret, though still manageable (Fig 1.1).

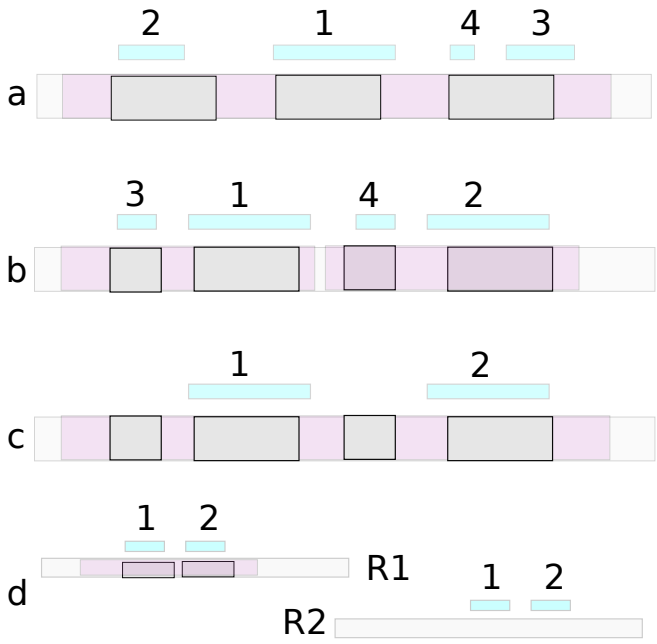

Figure 1.1: Scenarios for the BLAST results of a sequence and a genome sequence
Four possible scenarios are described in the text and illustrated here. In the above diagram the genome sequence is represented by the light-grey background rectangle. Over that a light-purple rectangle has been drawn to denote the gene region, and over that additional grey rectangles have been drawn to represent the coding regions. ESTs are represented above the genomic regions and the regions with similarity to the genome sequence are drawn in blue; regions lacking similarity to the genome sequence are not shown. The numbers above the similar regions represent the hypothetical HSP number of the match in the Hit. a) A relatively straight-forward sceneraio wherein the the EST matches to the exons of the gene. b) A more difficult scenario involving two tandemly duplicated genes which would most likely result in a single Hit containing 4 HSPs. In this case if one did not look closely at the BLAST report they might not notice that HSP 1 and 2, and 3 and 4 correspond to the same regions of the EST. c) A scenario wherein a single EST matches two exons in the gene, but skips one internal exon. This might result from the EST matching two duplicate elements in a single gene or because the internal exon is too short or contains low-complexity sequence that was filtered out by BLAST. Without access to the gene annotation, it is impossible to know for certain which case applies. d) A scenario wherein there are two strong Hits to different regions of the genome. This could be a result of matching a duplicated gene. In this example only one of the matches is to an annotated gene sequence and again, it would not be known without access to the annotation.

In a more complicated scenario, the gene may contain repetitive regions that cause the same region of the EST to match multiple regions in the gene. Without access to the genome annotation it would be difficult to know that one was dealing with a repetitive gene and not matching two separate genes in the same region of the genome. In another case, there might be exons in the gene that are not present in the EST. Without a representation of the alignment of the EST with the gene (or a detailed look at the annotation) this fact would not be readily apparent from the raw BLAST report, which would simply reveal a gap between the two HSPs that might be interpreted as an intron. In yet another case, the sequence may have strong similarity to two or more distinct regions of the genome. This could result from the presence of a highly conserved gene family in the model organism, or it could result from having sequenced a repetitive element (Fig 1.1).

The above difficulties arise from a lack of contextual information about the *Arabidopsis* genome sequences in the BLAST reports. The information needed to interpret the results is present in the *Arabidopsis* annotation, and representing the *Brassica* sequences relative to a marked-up representation of the genome provides the necessary context and gives users immediate access to the annotations. To this end the BioViz: *Brassica* / *Arabidopsis* Comparative Genome Browser was developed. While BioViz was only applied to the *Arabidopsis* / *Brassica* crop/model pair, it could be applied to comparative browsing relative to any model organism genome.

1.3 Solution

BioViz was developed as a web-based application in order to facilitate use by the scientific community (Fig 1.2). To provide a more powerful browsing experience than was available in existing web-based genome browsers [20, 18], the user interface and data representations were developed using the Scalable Vector Graphics (SVG) format [16]. *Brassica* sequences represented included expressed sequence tag (ESTs) sequences, whole genome shotgun (WGS) sequences and *Arabidopsis* sequences include activation tagged site (ATS) sequences, and serial analysis of gene expression (SAGE) tags. Representing the sequences in the context of the *Arabidopsis* genome sequences necessitated representations of the the *Arabidopsis* chromosomes and annotated features, specifically the *Arabidopsis* genes and splice forms.

The pairwise alignments displayed in BioViz were computed using BLAST and BLAT (the BLAST-Like Alignment Tool), and all the alignments are displayed in the context of the *Arabidopsis* genome. In addition to making the *Arabidopsis* annotations easily accessible, displaying the alignments in the context of the genome has the effect of "stacking" pairwise alignments with similarity to the same region, thereby relating the results of the individual pairwise alignments. This provides a result similar in nature to a MSA and allows the user to get a high-level impression of the relationship between all similar sequences without the need for a computationally expensive

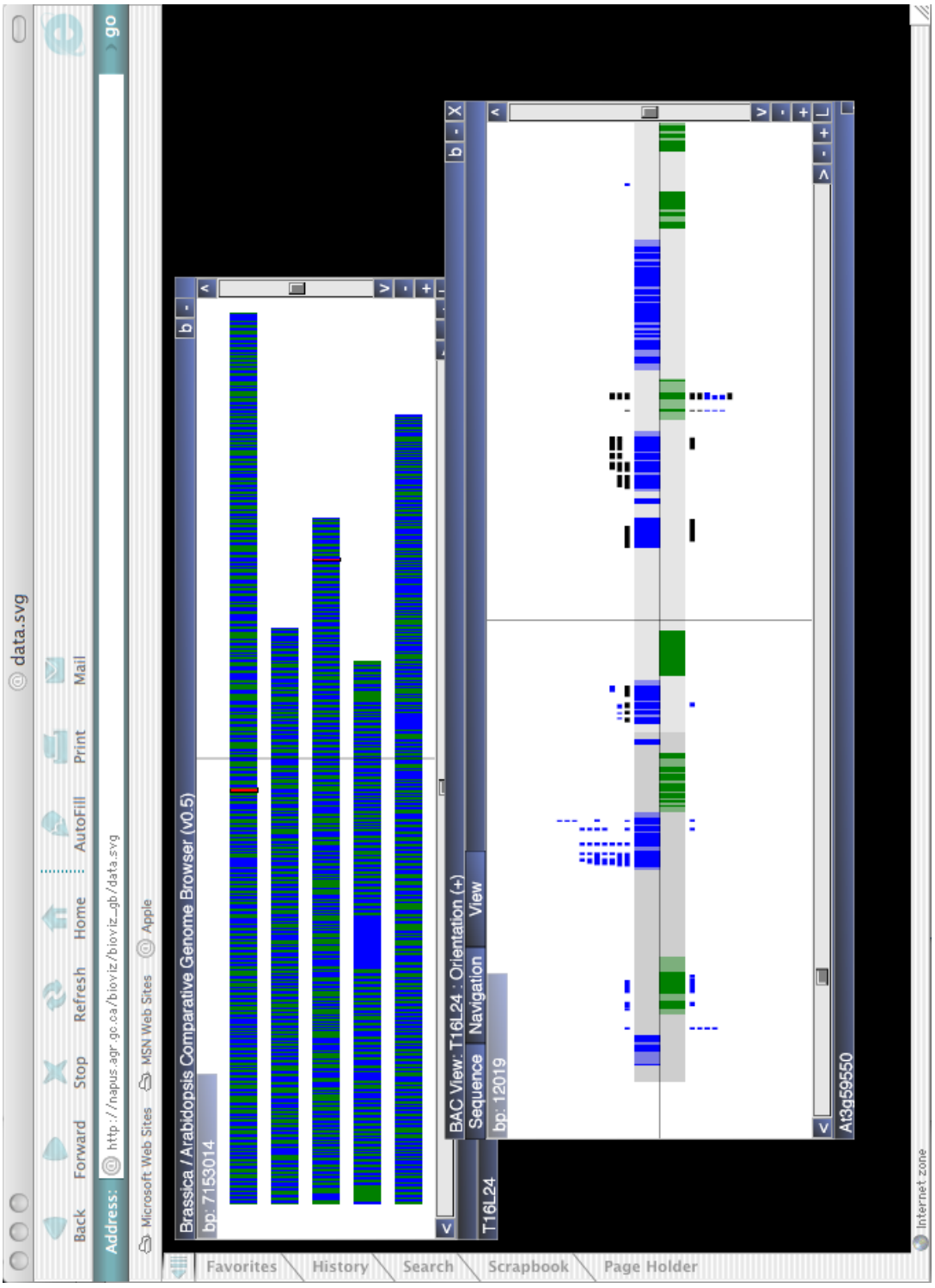

Figure 1.2: BioViz: Comparative Genome Browser
A screenshot of BioViz showing the initial chromosome display of the *Arabidopsis* genome and a view of the region containing BAC T16L24. AAFC EST sequences with homology in this region are stacked above and below the chromosome sequence. The view has been translated to the 5' most (left-most) end and zoomed in to show the structure of the genes and EST matches.

MSA for each set of sequences.

1.4 Related Work

At the time of development there were two other web-based genome browsers available (Ensembl [24] and Vista [20]), neither of which were available for the *Brassica* / *Arabidopsis* context. Since then other genome browsers have been developed, notably the Generic Model Organism Database (GMOD) GBrowse genome browser. However, the user interface developed and data representations used in BioViz are unique to BioViz. SVG enabled a more flexible user interface than is available in the other web-based genome browsers, and lessons learned in the development of BioViz are as relevant now as at the time of development. Futher, the emphasis on BioViz as a "comparative" genome browser is evident in the data representations and types of actions a user can perform from within BioViz and differentiates BioViz from these other browsers.

1.5 Impact

The end result attracted attention within both the *Brassica* research community where the representations provided by BioViz were used to estimate the gene copy number for *Arabidopsis* and *Brassica* genes in the sinapine biosynthetic pathway [3], the *Brassica* bioinformatics community where it was suggested that it could be used to complement an under development *Brassica* Ensembl database [38], the general bioinformatics community where it was cited as an example of the use of open standards in bioinformatics [57], and the SVG community for its novel application of SVG and the attempted creation of a reusable SVG UI widget set [35, 17]. The work resulted in three published papers and several poster presentations and demonstrations. The first two papers are related to the browser itself [34, 36] while the third is related to the user interface elements developed for the browser [31]. The content from the papers related to the browser has been extended and reworked for inclusion in this thesis with the application and data representations described in one chapter, and implementation details described in a second chapter. The third paper related to the user interface elements has been minimally modified and included to illustrate the impact of the work outside the bioinformatics community. Where appropriate, the material has been updated to reflect developments since the original publication.

1.6 Thesis Organization

This thesis consists of six chapters, including this introduction (Chapter 1):

- Chapter 2 provides background related to the BioViz application, including the underlying

concepts and implementation technologies.

- Chapter 3 introduces BioViz, including the objectives of the work, the data representations developed, the use of existing tools, and a comparison with other web-based genome browsers.
- Chapter 4 describes the implementation of BioViz making use of examples from the implementation to illustrate the key design and implementation principles.
- Chapter 5 illustrates the impact of the BioViz application outside the bioinformatics community.
- Chapter 6 summarizes the work completed as part of the thesis, highlights the key results and outlines future work.

Chapter 2

Background

2.1 Synopsis

This chapter begins with an introduction to sequence alignment[1], including heuristic local-alignment searches. It then provides background on additional key topics related to BioViz including EST (Expressed Sequence Tag) sequencing and comparative genomics. Finally, it introduces the technologies used in the implementation of BioViz including SVG, Perl and AJAX (Asynchronous JavaScript and XML). Readers already familiar with the above topics may wish to skip ahead and refer back to this chapter if necessary in the course of reading the subsequent chapters.

2.2 Sequence alignment

2.2.1 Biological basis

For most applications of sequence alignment the sequences are assumed to be evolutionarily related, which is to say homologous. Thus the objective is to find the regions that have been conserved through evolution and/or the regions that make one organism distinct from another. Homology is distinct from sequence similarity. While one might perform a sequence alignment and find that the sequences are similar, that does not necessarily mean that they are homologous. For instance, *Brassica napus* (commonly known as rutabaga, Swedish turnip, canola, rape) is thought to have formed within the last 2000 years through the hybridization of the *Brassica oleracea* (kale) and *Brassica rapa* (turnip) genomes [40, 44]. Consequently, one would expect the CBF (C-repeat/DRE-Binding Factor) genes in Figure 2.1 to be globally related, which is in fact the case for the *Brasica* and *Arabidopsis* sequences. The final *Secale cereale* (rye) sequence AF370730 is a more distant homologue included as an outlier. However, in other cases the sequences share biologically significant sequence-based features (regions of local similarity) despite their lack of global similarity. When aligning such sequences the objective is to identify the homologous features; however, this task may

[1] A significant portion of the material related to sequence alignment was developed for the author's comprehensive exam.

```
Sc:AF370730.1      GCA-GCCCGCGGGAGATCAAGGCAGCCGTCGCCGTCGCCGTCATCGCGTTCCAGCGGAAG
Bo:AF370731.1      ACCTGCCACAAGGATATCCAGAAGGCTGCTGCTGAAGCCGCATTGGCTTTTGAGGCTGAG
At:AF370732.1      ACCTGCGCCAAGGATATCCAGAAGGCTGCTGCTGAAGCCGCATTGGCTTTTGAGGCCGAG
Bn:AF499034.1      ACCTGCGCCAAGGATATCCAGAAGGCTGCTGCTGAAGCCGCATTGGCTTTTGAGGCCGAG
Bn:AF499033.1      ACATGCCCCAAGGAGATTCAGAAGGCGGCTGCTGAAGCCGC--------------------
Bn:AF499031.1      ACATGCCCCAAGGATATCCAGAAGGCGGCTGCTGAAGCCGC--------------------
Br:DQ022955.1      ACCTGCGCCAAGGATATCCAGAAGGCTGCTGCTGAAGCCGCATTGGCTTTTGAGGCCGAG
Br:DQ022954.1      ACATGCCCCAAGGATATCCAGAAGGCGGCTGCTGAAGCCGC--------------------
                    *  **     *** **  **   ** *  ** *  ****
```

Figure 2.1: Partial alignment of closely related CBF genes
CBF genes were selected from *Brassicas* (Bo, Bn, Br), *A. thaliana* (At) and *S. cereale* (Sc). The CBF genes from the *Brassicas* and *A. thaliana* are quite similar because the organisms are quite closely related. *S. cereale* is a more distant relative, and consequently the CBF gene contains less sequence similarity. Note that there appears to have been either a deletion or insertion event in three of the sequences relative to the others, but that the *S. cereale* sequence seems to contain entirely different sequence at that position which has still been forced into an alignment.

```
gi|154269265       ----------------------------ACTAATTGCTATGAT-----------------
gi|123977204       ---TCATGCAGATCTCCAATGAGTAT--GCTTTCTTCTGTAATCAAAAATGTGT------
gi|154707889       CCGTCAAGAACGTCACCTGTGAGAATGGGCTTCCGGCTGTGGTGAGCTGTGTGCCCGGCC
                                           **     ** *  *
```

Figure 2.2: Alignment of unrelated sequences
An alignment of three completely unrelated sequences. Note that there are short spurious matches between the sequences because the alignment algorithm will always produce an alignment regardless of the source or nature of the sequences.

be complicated by the large numbers of gaps, mismatches and spurious short matches within the unrelated regions of the sequences (Fig 2.2).

It is important that "similar" is understood to mean having similar biochemical properties rather than simple sequence similarity. From a biological perspective this means that an accurate sequence alignment aids in the identification of biologically relevant patterns and regions in the sequences. From a computational perspective it means that matches occur with a frequency based on the similarity of the two represented elements. It is this understanding that makes the alignment in Figure 2.3 meaningful, because, while the sequences have low sequence similarity they code for similar amino acid sequences, as can be seen from the protein alignment of the same region below the nucleotide alignment.

2.2.2 Brief history

Pairwise alignment

The concepts of local and global alignment come from the Needleman-Wunsch (NW) [41] and Smith-Waterman (SW) [51] pairwise alignment algorithms. These two mathematically exact sequence alignment algorithms were developed to align pairs of sequences, NW to perform global

```
AEL038C            GGCGTCG-----TTGACGTAGACGTCCGCGAGCGACA----TGAGG---CCCTGGCGG--
SPBC14F5.04c      AACGTCGAATCCATGATGGCCAAGGCCAAGAAGAACAACGTCGAGGTCTTCCTCCCCGTT
YCR012W           ATCGTTCCAAAGTTGATGGAAAAGGCCAAGGCCAAGGGTGTCGAAGTCGTCTTGCCAGTC
orf19.3651        AACGTTGAACACTTGGTTGAAAAAGCTAAGAAAAACAATGTTGAATTGATCTTGCCAGTT
                    ***        **      *   *   *    *       **      * *  * *

SPBC14F5.04c        N  V  E  S  M  M  A  K  A  K  K  N  N  V  E  V  F  L  P  V
orf19.3651          N  V  E  H  L  V  E  K  A  K  K  N  N  V  E  L  I  L  P  V
YCR012W             I  V  P  K  L  M  E  K  A  K  A  K  G  V  E  V  V  L  P  V
AEL038C             I  V  P  K  L  A  E  K  A  K  K  N  G  V  K  I  V  L  P  V
                       *        :        *  *  *     :  .  *  :  :  .  *  *  *
```

Figure 2.3: Alignment of distant sequences
An alignment of four pkg1 (phosphoglycerate Kinase) sequences from distantly related species: SPBC14F5.04c:*Schizosaccharomyces pombe*, YCR012W:*Saccharomyces cerevisiae*, AEL038C:*Ashbya gossypii*, orf19.3651:*Candida albicans*. At the protein level (bottom) these sequences contain substantially greater sequence similarity than at the nucleotide level (top).

sequence alignment (producing the optimal alignment of two sequences over their full length) and SW to perform local sequence alignment (producing the optimal alignment of two sub-regions of the sequences such that the two sub-regions provide the best possible score) (Fig 2.4).

SW and NW are based on similar recurrence relations, which can be represented as a 2-dimensional (2D) graph known as an edit-graph (Fig 2.5 a). In the edit-graph aligned characters are represented by a diagonal transition between nodes, while insertions and deletions are represented by a horizontal or vertical transition. Both algorithms use dynamic programming techniques to efficiently populate the graph with a simple recurrence relation (Fig 2.5 c shows the recurrence relation for a global alignment). The first entry in the recurrence relation calculates the score for a diagonal transition (i.e. match/mismatch). The second and third entries calculate the score for a horizontal and vertical transition respectively (i.e. a gap insertion into the sequence on the vertical or horizontal axis). The example in Figure 2.5 uses a constant (non-affine) gap penalty, though variable (affine) gap penalties can be used instead. After populating the edit graph a traceback step (Fig 2.5 b) can be used to determine the optimal alignment (Fig 2.5 d). The traceback here produces a global alignment, but it is possible to produce a local alignment or semi-global alignment by using a slightly different recurrence relation and altering the initialization of the matrix and varying the start and end points of the traceback.

Generalized pairwise alignment

A reasonable first attempt at multiple sequence alignment (MSA) would be to generalize one of NW or SW for use with 3 or more sequences. This can be accomplished by adding additional dimensions to the edit graph (Fig 2.6). However, the generalized algorithm requires $O(n^m)$ time for m sequences of length n, and even with the application of branch-and-bound techniques to reduce the search space [8], this approach is still intractable for most non-trivial applications.

```
########################################
# Program:  needle
#========================================
# Aligned_sequences: 2
# 1: EU024516.1
# 2: XM_001330739.1
# Matrix: EDNAFULL
# Gap_penalty: 5.0
# Extend_penalty: 1.0
#
# Length: 1474
# Identity:     329/1474 (22.3%)
# Similarity:   329/1474 (22.3%)
# Gaps:        1114/1474 (75.6%)
# Score: 767.0
#========================================

ATG---GCG-G----G-AAGAGC------------T-CAAAT----T---     21
|||   ||| |    | ||.|||           | |||||    |
ATGTTAGCGAGTTCAGCAACAGCAATGAAAGGATTTGCAAATAGTCTCAG     50

AA-GACAAG-C---AGCTCAGCA-----CTAA--TTGCTATG--------     51
|| || ||| |   ||||||.||     | ||  |||.||||
AATGA-AAGACTATAGCTCAACAGGTATC-AATTTTGATATGACAAAATG     98

-----ATTGC---TGATG----AG-G-AT--ACAGTAAC-T----GG---     77
     |||||   |.|||    || | ||  |||  ||| |    ||
CATAAATTGCCAATCATGTGTCAGAGCATGCACA--AACATCGCAGGTCA    146

--TCCGTTGC--ATCTGGAA-G-GTAT---------TG-A              393
  ||  .|||  ||||..|| | ||||         || |
GCTC--ATGCAGATCTCCAATGAGTATGCTTTCTTCTGTAATCAAAAATG    667

Remainder trimmed - Sequence 2 continues to end
```

```
########################################
# Program:  water
#========================================
# Aligned_sequences: 2
# 1: EU024516.1
# 2: XM_001330739.1
# Matrix: EDNAFULL
# Gap_penalty: 5.0
# Extend_penalty: 1.0
#
# Length: 677
# Identity:     326/677 (48.2%)
# Similarity:   326/677 (48.2%)
# Gaps:         320/677 (47.3%)
# Score: 770.0
#========================================

1   ATG---GCG-G----G-AAGAGC------------T-CAAAT----T---
    |||   ||| |    | ||.|||           | |||||    |
1   ATGTTAGCGAGTTCAGCAACAGCAATGAAAGGATTTGCAAATAGTCTCAG

22  AA-GACAAG-C---AGCTCAGCA-----CTAA--TTGCTATG--------
    || || ||| |   ||||||.||     | ||  |||.||||
51  AATGA-AAGACTATAGCTCAACAGGTATC-AATTTTGATATGACAAAATG

52  -----ATTGC---TGATG----AG-G-AT--ACAGTAAC-T----GG---
         |||||   |.|||    || | ||  |||  ||| |    ||
99  CATAAATTGCCAATCATGTGTCAGAGCATGCACA--AACATCGCAGGTCA

370 --TCCGTTGC--ATCTGGAA-G-GTAT    390
      ||  .|||  ||||..|| | ||||
620 GCTC--ATGCAGATCTCCAATGAGTAT    644
```

Figure 2.4: NW and SW alignment of two sequences
This figure shows the NW (left) and SW (right) alignment of two sequences as computed using the Emboss programs "needle" and "water" respectively. The alignments are effectively identical with the selected gap opening and extension penalties. However, the SW alignment is much shorter (677 bases vs. 1474 bases). The SW alignment stops at base 644 but the NW alignment continues even though one sequence has no more characters (to reduce the size of the image the NW alignment has been trimmed after character 667). The alignments were created with a standard distance matrix (EDNAFULL), and with a gap opening penalty of 5 and a gap extension penalty of 1.

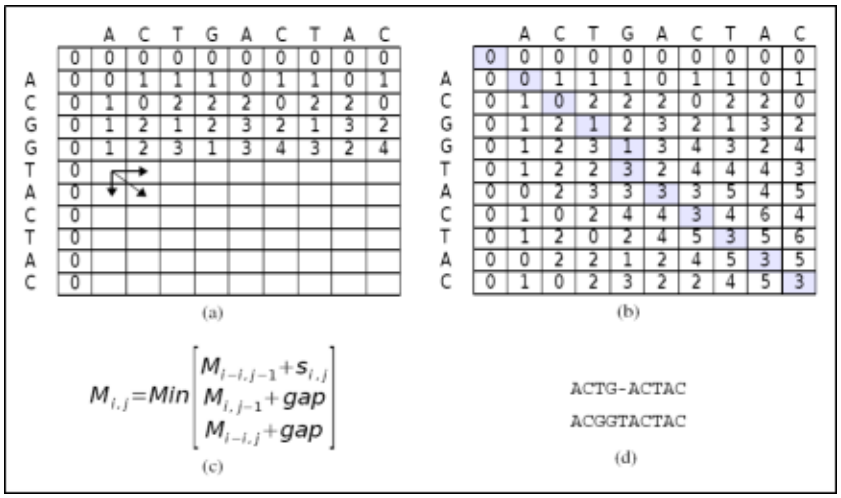

Figure 2.5: DP edit graph
The matrix representation of the DP relation used in the Needleman-Wunsch and Smith-Waterman algorithms. a) (top-left) The construction of the DP matrix used in the algorithm. b) (top-right) The traceback step (entries highlighted) which provides the optimal alignment. c) (bottom-left) The recurrence relation used with a non-affine gap penalty. d) (bottom-right) The resulting optimal alignment.

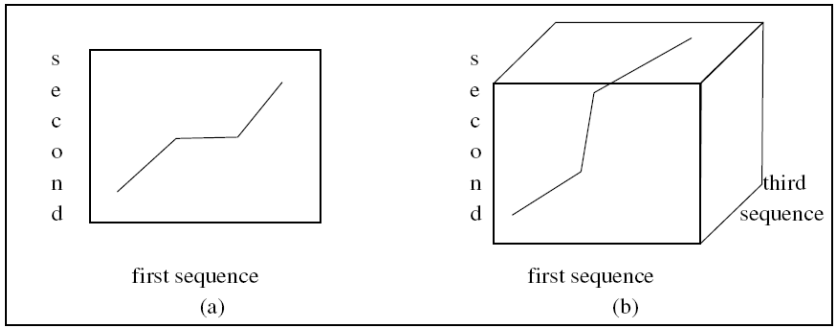

Figure 2.6: 2-D and 3-D edit graph
(a) shows the path through the 2D edit graph (2 sequences), while (b) shows the path through a 3-D edit graph (3 sequences). Note that the origin in these diagrams is in the bottom left rather than in the top right as in Figure 2.5 to make it easier to label the axes in part b.

Progressive multiple sequence alignment

The next approach to MSA, which gave rise to the current standard approach, was to build the MSA one pair of sequences (or alignments) at a time [15]. This progressive alignment is accomplished by aligning an initial pair of sequences, collapsing that alignment into a 1-Dimensional (collapsed) representation of the alignment and then aligning the 1D representation with another sequence or 1D alignment. The order of sequence alignment is generally determined by a guide tree – a bifurcating tree with internal nodes connecting the ever more distantly related sequences. The 1D representation of a MSA can be produced in a number of ways, but the two most common techniques involve reducing the MSA to a consensus sequence with the most likely base in each position or a profile that tracks the frequency of each base in a given column.

The major problems with the progressive approach are that it is sensitive to the input order of the sequences, and that it makes no guarantees regarding the optimality of the resultant alignment (Fig 2.7). The input order of the sequences may result in the introduction of gaps that are not appropriate in the context of subsequent sequences. However, following the "once a gap, always a gap" philosophy of Feng and Doolittle [15], these "erroneous" gap insertions are propagated through to the final alignment and may be worsened by the use of additional gaps to accommodate the initial error.

The possible improvements to the progressive alignment include varying the input order of the sequences and iteratively refining the alignment (effectively equivalent to non-stochastic iterative alignment below). The former is motivated by the experience that the pair-wise alignment order affects the resulting alignment quality, and the resulting belief that there is an input order likely to produce the best possible progressive alignment. The currently accepted approach is to minimize the sequence distance between each pair of aligned sequences, which should minimize the number

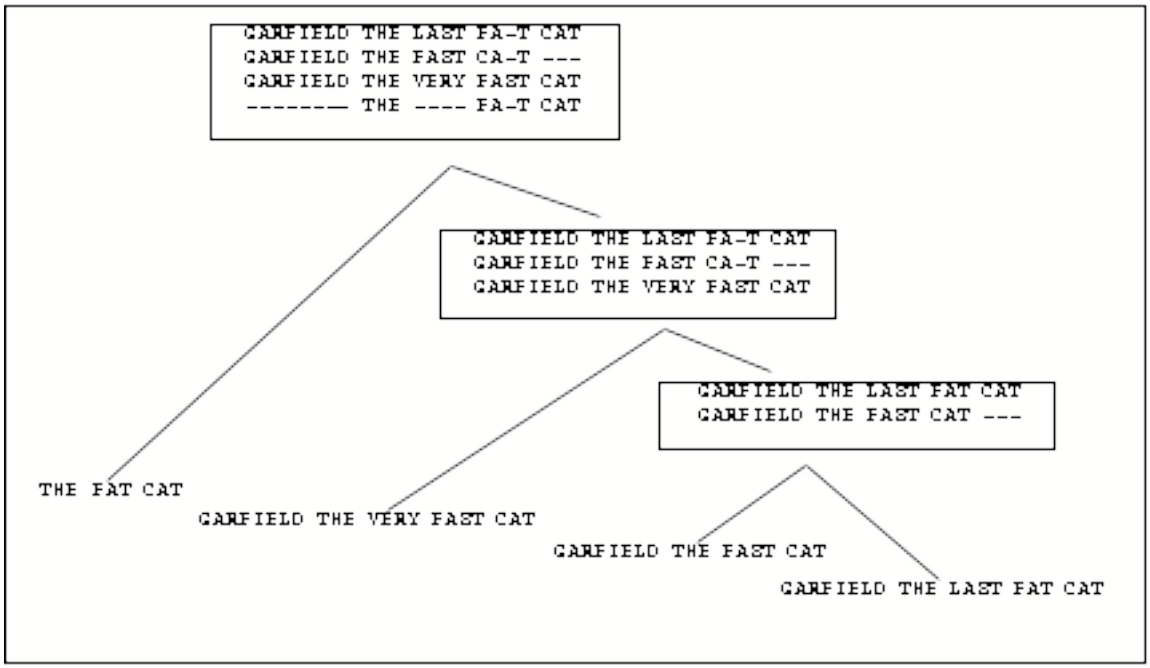

Figure 2.7: Illustration of initial gap selection

The above diagram illustrates how the progressive alignment strategy can introduce gaps which are likely not correct in light of subsequent sequences. In this case, the algorithm could have introduced an internal gap and aligned CAT with CAT, which would appear to be the correct decision considering subsequent sequences (alignment proceeds up from the bottom). Instead it choose a single position mismatch and a terminal gap because such gaps are often not penalized. Recreated from Notredame, 2002 [42].

of changes and thereby increase the accuracy of the alignment. The latter generally improves alignment quality, because the realignment of existing sequences may correct errors introduced in the early stage of the alignment. Historically it seems the refinement stage was often skipped because of the increased runtime, but the recent Muscle MSA implementation [14] performs several rounds of iterative improvement by default.

Iterative multiple sequence alignment

The non-stochastic iterative MSA algorithms work by extracting one or more sequences from the alignment, and then realigning the sequences with the alignment. This process is repeated until a constant alignment is achieved. While this often improves the quality of the alignment, the end result is still likely to be a local maximum rather than the optimal alignment. To increase the likelihood of achieving the optimal alignment, stochastic iterative alignment algorithms incorporate a degree of randomness into the solution. For example, SAGA is a stochastic iterative alignment algorithm based on genetic algorithms [43]. A genetic algorithm randomly rearranges a portion of the result after each iteration to see if the overall quality has improved, or at least not decreased below a defined threshold. This random rearrangement often allows the algorithm to break out of a local minima/maxima by moving to another area of the possible solution space. While such stochastic iterative alignment algorithms have been shown to produce good results, they are generally too slow and offer no guarantee of reaching the optimal alignment (or even convergence).

For a recent review on iterative multiple sequence alignment algorithms, see Wallace et al. [59].

2.2.3 Local alignment searches

A local alignment search algorithm finds non-spurious local alignments between a query sequence and a set of subject sequences. Performing a full dynamic programming search of the set of subject sequences is generally too time consuming, so heuristics are employed to speed the search. BLAST (Basic Local Alignment Search Tool) is the *de facto* standard tool for local alignment searches. It performs a seeded search against a pre-indexed collection of subject sequences (referred to as the BLAST database). As part of the search, all "seed" (short exact) matches are extended in both directions until the match score drops below a pre-defined threshold. A more detailed description is provided in Section 2.3.4.

2.3 Comparative genome browsing

A comparative genome browser should allow easy comparisons between two genomes, likely a fully annotated model organism and a more complex organism. To facilitate understanding of the more complex organism, the browser should provide access to the annotations associated with the model

organism. The following subsections provide background concepts related to comparative genome browsing.

2.3.1 Comparative genomics

Comparative genomics is the study of genome sequences relative to one another. It is a valuable research tool that significantly speeds biological discovery. The fundamental aspect of comparative genomics is the identification of similar and dissimilar regions in the genomes. Conserved regions present in multiple genomes are assumed to have come from a common ancestral sequence (and thus to be homologous), and so, at least historically, are likely to have had similar functions based on the principle that similar sequences provide similar function. However, the degree of conservation within the regions relative to the average similarity within conserved gene regions can be used to estimate the type of selection occurring within that region. If the region is experiencing stabilizing selection, which is to say that the regions have maintained more than average similarity over the course of time, then the regions likely still provide the same function in the two organisms. On the other hand, if the regions are experiencing positive selection, resulting in the sequences diverging more than the average, then the regions are likely responsible for providing different functions in the two organisms.

2.3.2 Genome composition

For the purposes of this thesis, the genome of an organism consists of one or more chromosomes. These molecules are composed of DNA and contain regions that result in the production of protein sequences. These gene regions can contain both coding (exonic) and non-coding regions (untranslated region, intronic), of which the coding regions contribute to the resulting protein sequence. Protein sequences are produced in two steps. A messenger RNA (mRNA) sequence is transcribed from the gene region of the chromosome. This molecule consists of the exonic regions of the gene; the intronic regions are removed from the molecule post-transcription. The sequence of the mRNA molecule is subsequently translated into a protein sequence, which folds in on itself to form the 3-dimensional structure.

2.3.3 EST sequencing

Where one does not have full genome sequences to compare, Expressed Sequence Tags (ESTs) provide a valuable resource for comparative genomics. ESTs are regions of expressed RNA (strictly speaking messenger RNA) sequenced from complimentary DNA (cDNA). Single stranded RNA sequences tend to be unstable and difficult to work with, so pools of complimentary DNA (cDNA) are created from the mRNA. The "tag" is typically the length of a sequencing run, which, with

the latest generation of Sanger-based capillary electrophoresis sequencing, tends to be on the order of 600-800 bp. ESTs are often assembled to form longer "contig" sequences based on overlapping shared sequences at the end of the EST. One might choose to compare ESTs with a reference genome (as was done in BioViz), or with ESTs produced from a related organism. Both approaches allow for the study of functional elements within the two organisms.

2.3.4 BLAST and BLAT

BLAST (Basic Local Alignment and Search Tool) [1] is a very common bioinformatics application, which is valuable in a comparative genomics context as a method for identifying similar sequence regions (recall Section 2.2.3). As mentioned in the introduction, BLAST takes one or more query sequences and searches for similar sequences in a set of "subject" sequences. To enable a rapid search, the set of subject sequences is pre-indexed. The basic algorithm works by identifying short matches, or "seeds" in the database, and then extending the matches until the quality of the match drops below a predefined threshold. The quality of a match is defined in terms of the number of paired, mismatched and unmatched characters in the alignment. An "E-Value", the probability that this is a random match, is provided for each match. More recent "gapped" BLAST algorithms [2] allow the insertion of gaps into the matches, which results in longer matches that would have otherwise been split into shorter matches. The standard BLAST report is a plain text output, an example of which is shown in Figure 2.8.

BLAT (Blast-like Alignment Tool) [26] is another heuristic local alignment search tool with additional features that make it potentially more useful than BLAST for aligning mRNA sequences with genomic DNA. BLAT attempts to group nearby hits into larger alignments, for example the matches for individual exons in a gene would be grouped into one larger match corresponding to the gene. It will also attempt to adjust match boundaries where possible if they correspond to canonical splice sites. It has been shown to be substantially faster than BLAST at performing the searches.

2.4 Technologies

Three core technologies were used in the development of BioViz: SVG to provide the user interface and content, Perl to generate the SVG content and handle data manipulation, and an AJAX style approach to data being exchange. SVG is an XML-based vector graphics format, while perl is a programming language commonly used for bioinformatics as it is well suited to string manipulation. Each of these technologies is briefly described in the following sections.

```
BLASTN 2.2.14 [May-07-2006]


Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs",  Nucleic Acids Res. 25:3389-3402.
Database: phytophthora_infestans_0.fasta
           3486 sequences; 173,338,388 total letters

Searching

Query= supercont0_72_of_Phytophthora_infestans_1_P1 p2 (CA)14 28 940
967
         (1154 letters)



                                                                 Score    E
Sequences producing significant alignments:                      (bits) Value

supercont_72 of Phytophthora infestans                             1029   0.0
supercont_3162 of Phytophthora infestans                            912   0.0
supercont_1489 of Phytophthora infestans                            880   0.0
supercont_725 of Phytophthora infestans                             866   0.0
supercont_2605 of Phytophthora infestans                            835   0.0
supercont_3220 of Phytophthora infestans                            801   0.0



>supercont_72 of Phytophthora infestans
          Length = 5101

 Score = 1029 bits (519), Expect = 0.0
 Identities = 537/546 (98%)
 Strand = Plus / Plus


Query: 609  ggtgacgcattcactcagtttctgtgcgcgcgcatttctcaatccgtaagacaaactact 668
            ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 968  ggtgacgcattcactcagtttctgtgcgcgcgcatttctcaatccgtaagacaaactact 1027


Query: 669  accgcaccctaactcagggttactcaatctgcctcagcagccgcagactacgtacgagct 728
            ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 1028 accgcaccctaactcagggttactcaatctgcctcagcagccgcagactacgtacgagct 1087


Query: 729  ggctgctgcagccactggtcagagcctgagcttaactactgacctcatcgctccccactc 788
            ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 1088 ggctgctgcagccactggtcagagcctgagcttaactactgacctcatcgctccccactc 1147


 Score =  969 bits (489), Expect = 0.0
 Identities = 509/519 (98%)
 Strand = Plus / Plus


Query: 1    cggggtctgataggcacctgcacttgtgtcactcagtttgtcaattgggttatattgacg 60
            ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 360  cggggtctgataggcacctgcacttgtgtcactcagtttgtcaattgggttatattgacg 419


Query: 61   aagcagcaacagatacgaagtcgccgagacacttaagctcctcaaagacgacgcagagca 120
            ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 420  aagcagcaacagatacgaagtcgccgagacacttaagctcctcaaagacgacgcagagca 479


Query: 121  gccggtggcgannnnnnnnnntaatcaacggcggagaggtgataaaacgtttggtggcgt 180
            ||||||||||          ||||||||||||||||||||||||||||||||||||||
Sbjct: 480  gccggtggcgagggggggggggtaatcaacggcggagaggtgataaaacgtttggtggcgt 539
```

```
>supercont_3162 of Phytophthora infestans
          Length = 59696

 Score =  912 bits (460), Expect = 0.0
 Identities = 504/519 (97%), Gaps = 2/519 (0%)
 Strand = Plus / Plus


Query: 1     cggggtctgataggcacctgcacttgtgtcactcagtttgtcaattgggttatattgacg 60
              ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 11029 cggggtctgataggcacctgcacttgtgtcactcagtttgtcaattgggttatattgacg 11088


Query: 61    aagcagcaacagatacgaagtcgccgagacacttaagctcctcaaagacgacgcagagca 120
              ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 11089 aagcagcaacagatacgaagtcgccgagacacttaagctcctcaaagacgacgcagagca 11148


Query: 121   gccggtggcgannnnnnnnnntaatcaacggcggagaggtgataaaacgtttggtggcgt 180
              ||||||||||          |||||||||| ||||||||||||||||||||||||||||
Sbjct: 11149 gccggtggcgaggggggggg-taatcaacggcgaagaggtgataaaacgtttggtggcgt 11207


 Score =  904 bits (456), Expect = 0.0
 Identities = 503/519 (96%), Gaps = 2/519 (0%)
 Strand = Plus / Plus


Query: 1     cggggtctgataggcacctgcacttgtgtcactcagtttgtcaattgggttatattgacg 60
              ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 10274 cggggtctgataggcacctgcacttgtgtcactcagtttgtcaattgggttatattgacg 10333


Query: 61    aagcagcaacagatacgaagtcgccgagacacttaagctcctcaaagacgacgcagagca 120
              ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 10334 aagcagcaacagatacgaagtcgccgagacacttaagctcctcaaagacgacgcagagca 10393


Query: 121   gccggtggcgannnnnnnnnntaatcaacggcggagaggtgataaaacgtttggtggcgt 180
              ||||||||||          |||||||||| ||||||||||||||||||||||||||||
Sbjct: 10394 gccggtggcgaggggggggg-taatcaacggcgaagaggtgataaaacgtttggtggcgt 10452
```

Figure 2.8: Sample BLAST report

A BLAST report consists of a preamble, a summary of the Hits in the report and the details of each Hit. Each Hit provides the details of the alignment between one query-subject pair. Within each Hit there are one or more HSPs (High Scoring Pairs), which are regions of local similarity between the two sequences. The above example provides the preamble of the BLASTN (nucleotide-nucleotide) report. The first query sequence is 1154 letters and resulted in 6 hits. The E-Value of these hits (0) indicates that there is little statistical possibility that these are random matches. This example includes fragments of the output for two Hits. Each Hit begins with the ">supercontig" tag (this is the name of the subject sequence), and lists the details of the match. Also included is a fragment of the second HSP for each Hit. The HSPs begin with the second "Score = " block. The above report was produced as part of a marker identification analysis undertaken in the fungus *Phytophthora infestans* [32].

2.4.1 Representation

XML

XML (Extensible Markup Language) is a human readable, text-based, structured format for encoding information [6]. It has a format similar to HTML (Hyper Text Markup Language) in that data and other elements can be nested inside a parent element, and elements can have attributes that modify the element. However, unlike HTML which is specific to the creation of web pages, XML is a generic document format. The set of possible elements in an XML document is virtually unlimited, and a formal grammar, called a schema or data type document (DTD), can be defined to ensure that the final XML document is valid. The basic structure of an XML document is shown is Figure 2.9.

Vector vs. Raster Graphics

Vector graphics are defined by mathematical formulas describing the desired shapes. Raster graphics (or bitmaps) consist of a grid of pixels which can be set to a desired color and used to represent the image. Raster graphics are generally more useful for photographs because it is easier to break the captured image down into discreet pieces than it is to identify, define and style individual shapes within the image. On the other hand, Vector graphics are often created by and used in graphic art programs and printing because they can be infinitely scaled and otherwise manipulated through the application of transformations (mathematical descriptions of the desired change). Rather than coloring individual pixels, as in a raster graphics based graphics package, one draws basic shapes and paths. The simple shapes used for data representations in BioViz were very simple to create in this format, and the ability to infinitely scale a vector graphic was exploited in BioViz to allow the user to zoom in on features of interest without generating a new image on the server.

Scalable Vector Graphics

SVG [16] is an XML-based vector graphics format which offers interactivity via ECMAScript (i.e. JavaScript) [13] and Synchronized Multimedia Integration Language (SMIL) animation [7]. The SVG and SMIL specifications is defined by the World Wide Web Consortium (W3C). ECMAScript is a scripting language specification from Ecma International (Ecma). The specification treats shapes as first class citizens that can be grouped and easily manipulated through the application of transformations and stylesheets. The XML format coupled with the interactive nature of the graphics has resulted in considerable interest in the use of SVG for web application development [48, 50, 36], especially in the field of cartography [28]. The current SVG specification is version 1.2. However, BioViz was developed while the 1.1 specification was in development and does not take advantage of the functionality that became available in the 1.2 specification.

```
<?xml version="1.0" standalone="yes"?>

<svg xmlns="http://www.w3.org/2000/svg">

    <rect x="0" y="0" width="640" height="480" fill="blue"/>
    <g transform="translate(200,200) scale(.5)">
        <rect x="0" y="0" width="200" height="200" fill="white"/>
        <circle cx="100" cy="100" r="50" fill="red"/>
    </g>

</svg>
```

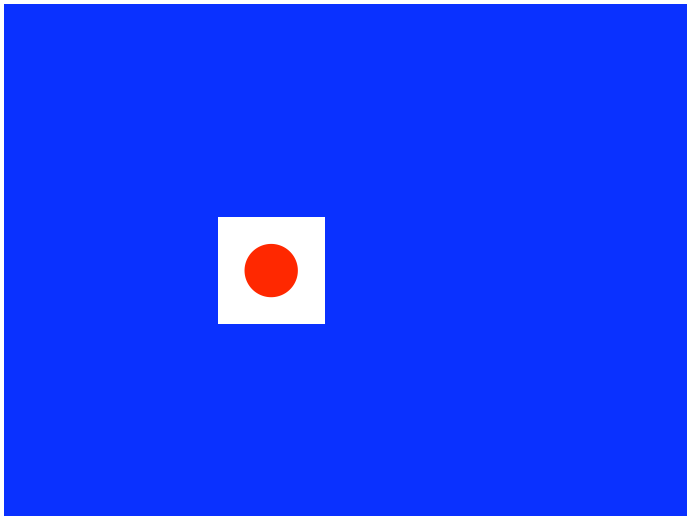

Figure 2.9: Sample XML Document
(Top) The first line declares this to be an xml document, while the second line begins the declaration of an SVG element containing a blue rectangle with a white rectangle containing a red circle drawn on top of it. An element in an XML document begins with an opening tag and ends with a closing tag. Tags begin with a "<" character and end with a ">" character. An element may be declared using an opening and closing tag e.g. "<g> ... </g>" or a single closing tag e.g. "<rect ... / >". The "/" character marks a terminal tag. All tags may contain attributes and elements may be nested elements. As an example the "circle" element contains 4 attributes: "cx" and "cy" provide the x and y coordinate of the center of the circle, "r" defines the radius of the circle and "fill" specifies the color of the circle. It is defined using a single tag nested in a parent "g" (group) element. (Bottom) The resulting SVG image.

```
<?xml version="1.0" standalone="yes"?>
<svg xmlns="http://www.w3.org/2000/svg">

    <text x="20" y="20">Hello World</text>

</svg>
```

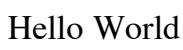

Figure 2.10: "Hello World" SVG example
(Top) A "Hello World" SVG document, which displays the text "Hello World". (Bottom) The rendered SVG Document.

```
<?xml version="1.0" standalone="yes"?>
<svg xmlns="http://www.w3.org/2000/svg">

    <rect x="0" y="0" width="120" height="30" fill="blue"/>
    <text x="20" y="20" fill="orange">Hello World</text>

</svg>
```

Figure 2.11: Colored "Hello World" SVG example
(Top) By adding a fill attribute to text element we can easily change the colors in the image. (Bottom) The rendered image with a color rectangle as a background.

The specification provides a simple, human-readable syntax for defining vector graphics. A simple "Hello World" example is provided in Figure 2.10. This example contains a single "text" element containing the text "Hello World". Figure 2.11 adds a "fill" attribute to the the text element to change its color and includes a blue rectangle in the background. Interactive graphics can be created by attaching "event listeners" to elements within the SVG Document. These can take the form of an "onload" listener, which will execute when the Document is loaded, or various "on[mouse*]" listeners which are executed in response to different "mouse*" events (e.g. the onclick event listener would be executed when the user clicks the mouse on the element). Figure 2.12 describes a SVG document with an "onclick" event listener that modifies the SVG Document when the user clicks on the rectangle. This is a trivial demonstration of the interactivity available in SVG, however, when used with the ability to retrieve content from the server and dynamically update the SVG Document (i.e. AJAX), the interactive features enable a very compelling browsing experience.

2.4.2 Data handling and transfer

Perl

The server-side of BioViz is implemented in Perl. Perl is commonly used for bioinformatics applications because of the ease with which strings can be manipulated. Consequently, there are many bioinformatics tools and source libraries available in Perl. Further, Perl facilitates the development of applications implementing the CGI (Common Gateway Interface) protocol, which is a standard for providing dynamic web content. The above, together with the existence of a Perl module for the generation of SVG content, made Perl a good choice for the server-side implementation of BioViz.

SVG.pm

The SVG perl module (SVG.pm) facilitates the creation of SVG content from within Perl. It defines a set of methods enabling the simple construction of an SVG Document and and a method for writing the result as an SVG format file. Figure 2.13 presents an example of the use of SVG.pm to generate the "Hello World" example presented in Figure 2.10.

BioPerl

There are a number of open-source bioinformatics libraries [39], for instance, BioJava [23], and BioPerl [53]. However, the BioPerl library tends to be the most actively developed and extensive library, presumably as a result of the popularity of Perl for bioinformatics applications. The greatest value to BioViz from BioPerl came in the form of parsers for commonly used bioinformatics file formats. However, using BioPerl one might also develop a flexible SVG-based interface to display

```
<?xml version="1.0" standalone="yes"?>
<svg xmlns="http://www.w3.org/2000/svg" width="140" height="80">

    <g transform="translate(5, 15)" text-anchor="middle">
        <rect id="button"
             x="0" y="0" width="70" height="25" fill="red"
             onclick='document.getElementById("target")
                             .setAttribute("fill", "blue")' />
        <text
            x="35" y="15" text-anchor="middle"
            >Click Here</text>

        <circle id="target"
            cx="100" cy="10" r="20" fill="white" stroke="black"/>

    </g>

</svg>
```

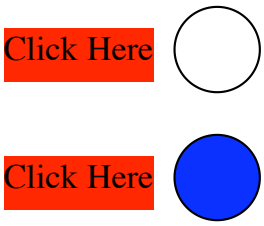

Figure 2.12: Interactive Hello World example
(Top) The SVG Document defines a "140 x 80" image containing a "red" rectangle with the text "Click Here", which has an "onclick" event listener that changes the color of the circle when the user clicks on it. It does this by setting the "fill" attribute of the "circle" element to "blue". The example also demonstrates use of the "id" attribute of an element in an XML document, which allows a given element to be referenced directly. (Bottom) The image before and after the user clicks on the rectangle.

```
#! /usr/bin/perl -w

use SVG;

my $svg = new SVG(
            width=>"640",
            height=>"480"
        );

my $text = $svg->text(
            x=>"320",
            y=>"240",
            "text-anchor"=>"middle"
    )->cdata("Hello World");

print $svg->xmlify();
```

Figure 2.13: Generate "Hello World" example using SVG.pm
The Perl code above generates the "Hello World" example using SVG.pm. The SVG Document is created using the "new SVG" call. The SVG Text element is created by calling the "text" method on the SVG Document reference ($svg). The XML document is created by calling the "xmlify" method.

information available in GBrowse.

AJAX

AJAX (Asynchronous JavaScript and XML) is a term that was coined to describe interactive web applications that asynchronously retrieve data from a server and use JavaScript to update the currently displayed document. Common examples include GMail (Google Mail) and Google Maps. BioViz is a prime example of such an application, though it did not use the XMLHttpRequest interface for client-server communication (the current standard approach for client server communication in AJAX) because it was not defined at the time BioViz was implemented. Instead, BioViz utilized functionality provided within the Adobe SVG Viewer to send HTTP POSTs to the server and asynchronously receive a result via a callback function, which was used to update the SVG document. A POST is a standard HTTP message used to send data from the client to the server, for instance, it is commonly used to submit entries in forms. XMLHttpRequest is a newer programming interface allowing XML (or other text based data) to be transferred between a client and a web server.

Chapter 3

BioViz: Description

3.1 Synopsis

This chapter describes the *Brassica* / *Arabidopsis* Comparative Genome Browser (BioViz), introduces the goals that motivated its development, and provides a comparison to other web-based genome browsers[1].

3.2 Background

The *Brassica* / *Arabidopsis* Genomics Initiative (BAGI) [29] is a project at the Saskatoon Research Centre of Agriculture and Agri-food Canada (AAFC). The project aims to develop genomic and genetic resources to expedite the characterization of gene function in *Brassica* and *Arabidopsis*. To this end, the initiative developed a number of biological resources, including 3' and 5' ESTs from *B. napus* cDNA clones. The *Brassica* / *Arabidopsis* Comparative Genome Browser (BioViz) was developed to provide context for the sequence information derived from these resources by displaying *Brassica* sequences relative to homologous regions of the *Arabidopsis* genome. This imparts contextual information on the *Brassica* sequences through access to the *Arabidopsis* annotation, as well as providing a loose clustering of *Brassica* multi-gene families relative to their *Arabidopsis* homologue. Currently, there are more than 70,000 3' and 5' *Brassica napus* EST sequences displayed in the public version of the browser, and in the future information related to expression information from microarray and Serial Analysis of Gene Expression (SAGE) studies as well as publicly available *Brassica* sequences may be included.

Arabidopsis thaliana is the model organism for plant genomics research most closely related to the *Brassica* genus, which contains many agriculturally important species. The genome sequence for this plant was completed in 2000 [45], and it consists of DNA sequence information organized into 5 chromosomes, each of which is up to 30 million base pairs in length and contains up to

[1]The material is derived from two early papers. The first, "BioViz: Genome Viewer, Development of an SVG GUI for the visualization of genome data" was a conference paper including a presentation at SVG Open 2002 [34], and the second "The Brassica / Arabidopsis Comparative Genome Browser, A novel approach to genome browsing" was published in the Korean Journal of Biotechnology 2003 [36] following two invited presentations related to the browser in Korea. The material has been updated to reflect developments since the initial papers were published

7,000 distinct genes distributed along the length of the chromosome. BioViz used the *Arabidopsis* genome sequence and annotations developed by The Institute for Genomic Research (TIGR), which is currently a part of the new J. Craig Venter Institute (JCVI). The initial version of the browser used annotations from TIGR released on August 10, 2001, which contained annotations for 25,617 genes. TIGR's ongoing annotation efforts [61] came to an end in January 29, 2004 with release 5, which is the latest version of the *Arabidopsis* genome data included in the browser. Release 5 contains annotations for 26,207 protein-coding genes and 3,786 pseudogenes genes of which 22,150 of the annotations are supported by sequence similarity to at least a plant EST [56].

3.2.1 Objectives and scope

The objective in developing a comparative genome browser was to help users visualize the relationship between *B. napus* ESTs (or other sequence-based features) and the *A. thaliana* model genome. To capitalize on the community effort to annotate *Arabidopsis*, the browser had to allow access to the *Arabidopsis* annotations, which required that it be a functional genome browser for *Arabidopsis*. The viewer was not intended to be an annotation tool, and consequently there was no requirement to update the underlying dataset (neither the *Arabidopsis* genome annotations nor the *Brassica* sequence annotations).

It was desired that the viewer be a web-application for two reasons. Firstly, the web provides a venue for advertising the availability and extent of the biological resources developed for the project. Secondly, making the data accessible online ensures the viewer and associated data can be easily accessed by the *Brassica* research community.

3.2.2 Existing tools

There are standard bioinformatics tools available, such as BLAST [1], which will find regions of similarity between given sequences. Unfortunately the plain text report produced by such standard tools is not always the most informative way to visualize the relationship. This is especially true when trying to visualize alignments over a large region such as: a BAC (Bacterial Artificial Chromosome), average size 100,000 bp [5]; a chromosome, average size 20-30 Mbp; or the whole genome. BioViz was developed to assist in visualizing such relationships.

BioViz allows us to efficiently display the alignment of many ESTs over a large genomic region such as a BAC (Fig 3.4). Regions of similarity between the *Brassica* sequences and the BACs in the *A. thaliana* genome were determined using NCBI BLAST and hits with an E-value greater than 10^{-4} were stored in a MySQL database. Using this information, the browser allows the user to view all ESTs which have similarity to a given BAC, to see how different ESTs align with the predicted *A. thaliana* genes, and to find homologous regions of *Arabidopsis* relative to specific ESTs. However, displaying the alignment relative to the genome adds context to the results that is not

otherwise present in the raw BLAST output.

3.2.3 Other genome browsers

BioViz was not the only genome browser available at the time of development[24, 54, 11, 27]. However, the existing browsers all had two main drawbacks which BioViz sought to overcome - frequent page reloads and fixed displays. These characteristics result from the use of server generated bitmaps to present the information and the application of traditional HTML layout techniques (with one exception). The page reloads occur quite frequently because a new image must be generated on the server each time the user wants to change the current view (by zooming in or out, looking at another region of the genome, or requesting supplementry information). This page reload is functional, however, it can be distracting – especially when communicating with a server via a slow network connection or a server which is under heavy load and slow to return results. The static layout is a result of the use of HTML tables to perform the layout. Like the page reload, it is functional, but it limits the ways in which users can customize their view of the data (Fig 3.1).

As an example, the Ensembl Genome Browser [24] provided a wealth of information in a straightforward manner. However, the data was presented in bitmap format so a page reload was required to retrieve a new bitmap each time the user wanted a different view of the data. The user could customize the bitmaps returned from the server using a series of combo boxes below the display area, but the interface itself was quite static with few-if-any possible customizations. The GBrowse [54] and USCS Genome Browser [27] interfaces were effectively equivalent to the Ensembl interface. The Vista Genome Browser [11] took a different approach.

Vista was created to view the results of the Berkeley Genome Pipeline for Human / Mouse genome comparisons. This browser used a Java Applet and relied on Java Servlets (effectively server-side applets) to update the display. It provided a graph showing regions of similarity between the mouse genome and human genome. By implementing the browser as a Java applet rather than using HTML, the developers avoided the page reloads common in other web-based genome browsers. Consequently, when compared to browsers such as the Ensembl Genome Browser the data refreshes were less noticeable and the browser felt more fluid. However, Vista provided the user with a static display, which is typical of most web-applications. For additional information related to the sequences, Vista linked to a version of the UCSC Genome Browser [27]. Another second significant difference between Vista and the other existing genome browsers was its focus on the alignment of whole genome sequences where the other browsers focus on showing short sequences in the context of a genome.

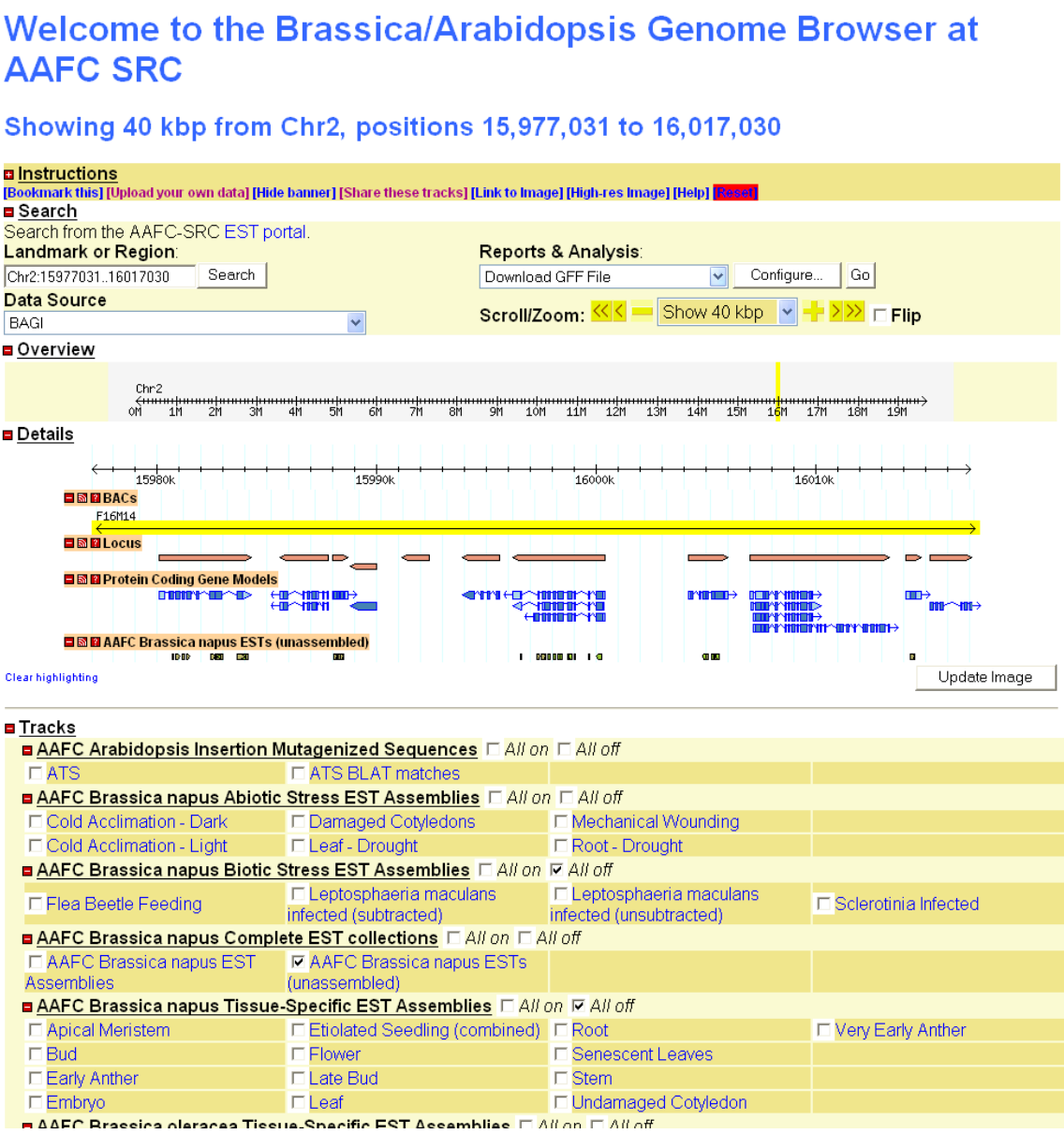

Figure 3.1: GBrowse "BAC View"

While GBrowse does not technically have a "BAC View", this screenshot shows the BAC which is displayed in the BioViz "BAC View" screenshot. This screenshot is taken from the GBrowse installation at Agriculture and Agri-food Canada in Saskatoon. The static layout is typical of the web-based genome browsers, and the configuration options are presented as a series of checkboxes below the graphic. The graphic presents an overview which shows the displayed region in the context of the chromosome. The Details area presents a bar for scale and represents chromosome-based features below it. For this screenshot the BACs, LOCUS, Protein Coding Gene Models and AAFC *Brassica napus* ESTs (unassembled) tracks were enabled.

3.3 Results

To satisfy the above objectives, the *Brassica* / *Arabidopsis* Comparative Genome Browser (BioViz) was developed. BioViz is a client-server application that enables the efficient display of similarities between many short *Brassica* sequences and the *Arabidopsis* genome (Fig 1.2). It allows the user to view all sequences with similarity to a given region of the *Arabidopsis* chromosome (Fig 3.3), to see how different ESTs align with the annotated *A. thaliana* genes (Fig 3.4), to search for named *Brassica* or *Arabidopsis* sequences, to access the annotation associated with the *Brassica* and *Arabidopsis* sequences and to perform on-the-fly BLAST searches for sequences within the database (Fig 3.6).

The client-side relies on a novel SVG-based Custom GUI library (CGUI) to provide an intuitive, easy-to-use, web-accessible front-end for this data. CGUI was written using JavaScript objects, and contains basic objects to facilitate the construction of a GUI. Windows can be opened, closed, and moved inside the GUI. The contents of a window can be scaled and translated independently from other windows which provides the user with a very flexible interface. Client-side scripting allows the user to create and dismiss views and request supplemental data from the server which allows them to see as much or as little information as they would like at one time. Asynchronous data retrieval eliminates the distracting page reloads common in other web-based genome browsers and allows the user to continue exploring the genome while additional data is loading.

The server-side is implemented in Perl and responsible for retrieving data and rendering it as SVG. Regions of local similarity between the *Brassica* sequences and the *Arabidopsis* genome were identified using BLAST and BLAT (c.f. 2.3.4) and the results were stored in a MySQL database. The genome data itself resides in a mix of XML files, flat-files and relational databases on the server (Fig 3.2).

Implementation details are provided in Chapter 4.

3.3.1 Data representations

BioViz was implemented to allow a "drill-down" style access to the displayed information. The application opens with a representation of the 5 *Arabidopsis* chromosomes and the user is able to navigate, via a few intermediate steps, to a more detailed view of the individual gene or ESTs.

Chromosomes

The *Arabidopsis* chromosomes are displayed as long rectangles with a constant height and a width (in pixels) equal to the number of nucleotides in each chromosome. The chromosomes are aligned on the left-hand side of the display and scaled to fit the width of the containing window (Fig 3.3). Experimental representations had a rectangular mask with rounded corners applied to give

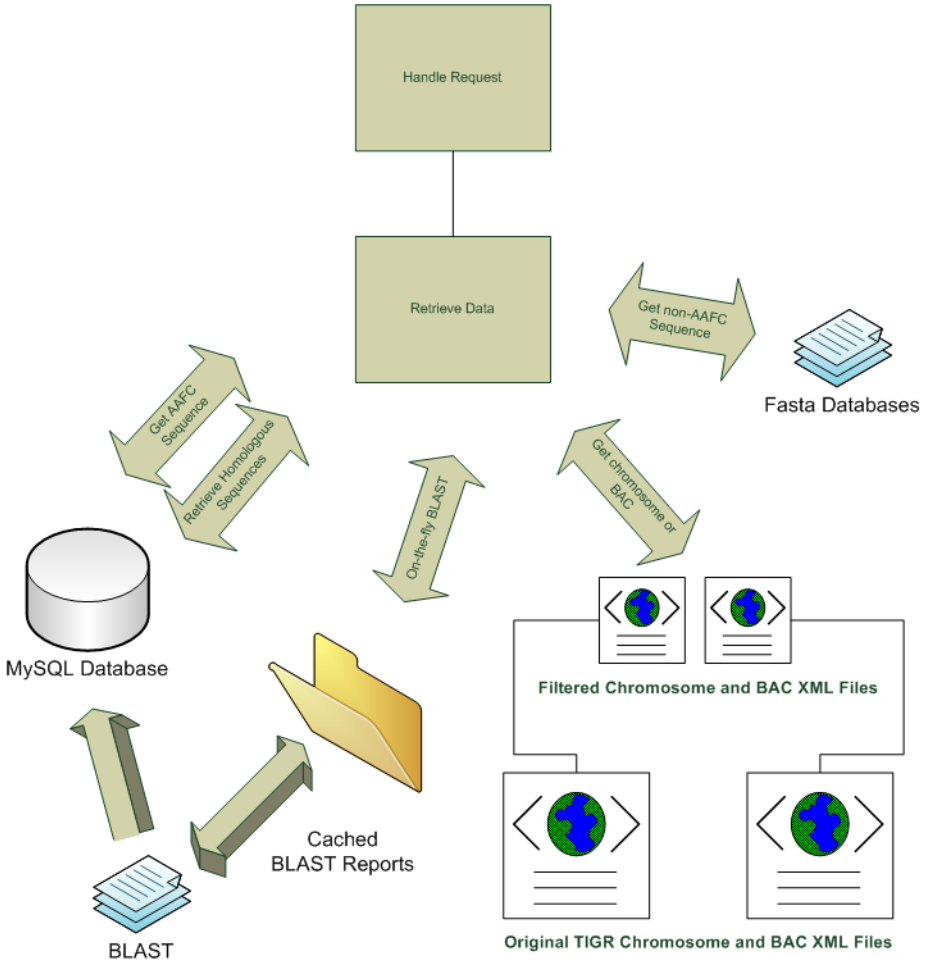

Figure 3.2: BioViz Data Flow

The data displayed in BioViz comes from a number of sources. The chromosome representations are created from an XML file, which contains a subset of the information contained in the TIGR chromosome.xml files for efficiency. The BAC representations are created from the TIGR BAC XML files, also filtered to reduce the size of the XML file. Sequence similarity information comes from a mysql database, on-the-fly BLAST report(s), or cached on-the-fly BLAST report(s) depending on the request. Sequences are extracted from one of several fasta format databases or the mysql data database depending on the request. All data sources are configurable on a per-installation basis. There are a number of pre-defined data access modules, or the local DBA can code custom data access modules.

the chromosomes their more usual "rounded" appearance, with a "pinched" region marking the chromosome's centromere, however this never made it into the released version of the browser. The centromere is the location where the two chromatids making up the chromosome join during the cellular reproductive cycle.

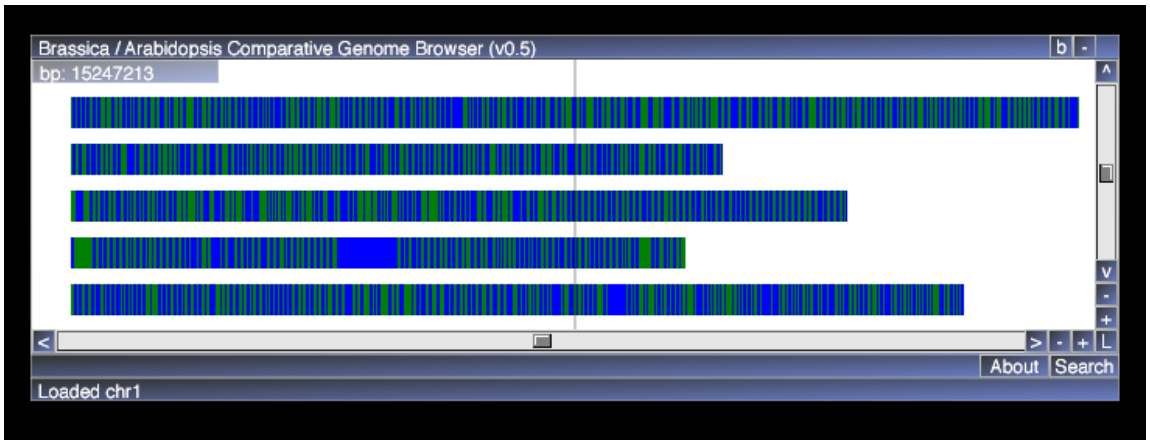

Figure 3.3: BioViz Chromosome View
The initial view presented to the user in BioViz is the "chromosome" view, which presents the 5 *Arabidopsis* chromosomes laid out horizontally. Chromosome 1 on the top through chromosome 5 on the bottom. Using the +/− buttons at the base of the window the user can zoom in or out on the chromosomes, and they can use the side scrollbars to move around. The line at the middle of the display follows the mouse and the position of the mouse relative to the chromosomes (in bp) is displayed in the top-left text area.

BACs

The *Arabidopsis* chromosomes were sequenced in overlapping Bacterial Artificial Chromosomes (BACs), and the BACs are marked on the chromosomes as alternating green-blue rectangles. In BioViz, the BACs provide the first level of "drill-down". The user can click on a given BAC and a new view will open displaying the corresponding region of the chromosome. Represented on that region are all the annotated genes, as well as the regions of overlap with the neighbouring BACs (Fig 3.4a).

Genes and gene structure

The genes in BioViz are shown within the rectangle representing the chromosome. Genes may be annotated as occurring on the $5'$ or $3'$ strand of the chromosome. Genes occurring on the $5'$ strand are drawn on the top of the chromosome and genes occurring on the $3'$ strand are drawn on the bottom of the chromosome. The genes are typically made up of coding and non-coding regions, the order of which is referred to as the gene structure. In BioViz, a semi-transparent rectangle is drawn

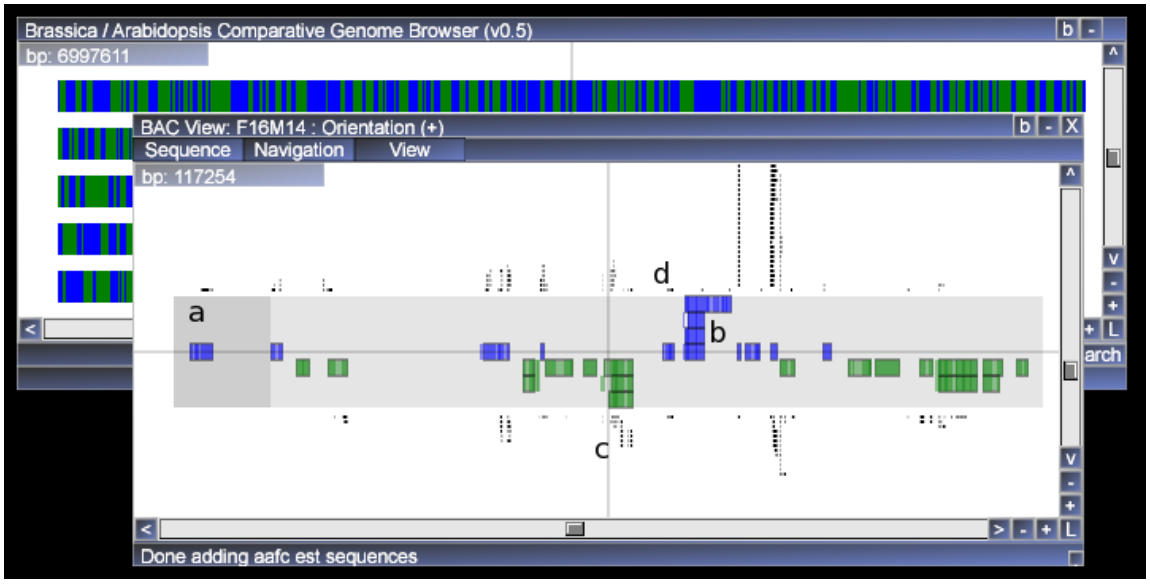

Figure 3.4: BioViz BAC View
The "BAC" provides the first level of drill-down in BioViz. At the BAC level the *Arabidopsis* genes and *Brassica* sequences with homology to *Arabidopsis* are displayed. a) The dark-grey area marks the overlap with the BACs 5′ neighbour in the chromosome assembly. b) A gene with multiple annotated gene structures. c) A set of ESTs having similarity to an *Arabidopsis* gene – note that the alignment closest to the genome has the lowest E-value. d) An EST having similarity to the *Arabidopsis* gene in the reverse orientation.

to represent the entire gene region (including non-coding regions at the 5′ and 3′ ends), and then additional semi-transparent rectangles are drawn overtop of this rectangle to mark the exons. In BioViz, where a gene has multiple annotated gene structures, the different versions appear stacked on top of each other (Fig 3.5).

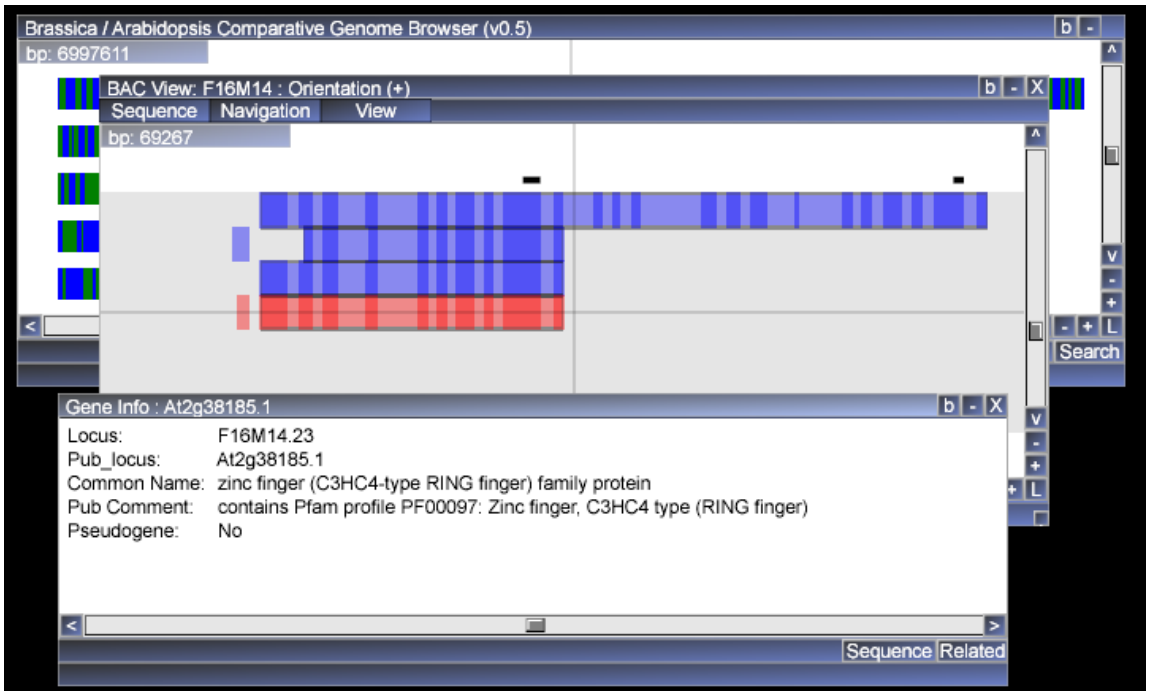

Figure 3.5: BioViz Gene View
From the BAC View, the user can zoom in to see the structure of genes and the relationship with related sequences. The user can also drill-down further to access the annotation associated with a given gene. From here the user can drill-down one further level and access the sequence associated with the gene (not shown), or search for other related sequences (not shown) using the buttons in the bottom-right.

Alignments

The alignments displayed in BioViz have been computed relative to the *Arabiodpsis* BAC sequences. Alignments are displayed relative to the BACs rather than the chromosomes because it was felt that there would be little information gained by displaying them at the chromosome level of resolution. However, the hits could be displayed relative to the chromosomes by converting the coordinates of the BAC-based hits to chromosome-based coordinates. Each BAC occurs at some position on the chromosome and each hit starts at a position on the BAC. To convert the alignment location to chromosome-based coordinates simply requires adding the 5′ most coordinate of the BAC to the alignment position.

Alignments are represented by drawing a fixed height rectangle with a width equal to the length

of the HSP for each HSP in a given BLAST Hit and an x coordinate corresponding to the start of the Hit relative to the subject sequence (Figure 2.8). Alignment with the BACs can be computed for multiple sets of query sequences, and each set of alignments can be displayed one at a time in the BAC view. Because all Hits in a set are displayed at the same time, it is necessary to adjust the y value for each Hit so that the Hits do not overlap. In BioViz, the "best" Hit (as determined by the E-value) is drawn closest to the BAC (Fig 3.4c).

Hits may be found in either the "+" or "−" orientation (that is, the subject and query sequence match in the same orientation or in the reverse orientation). Where a match occurs in the same orientation it is drawn on the side of the chromosome where the gene occurs. Where it occurs in the reverse orientation, it is drawn opposite the gene (Fig 3.4d).

BLAST results

BioViz allows the user to perform an internal BLAST against sequences in the underlying database. For instance, a user interested in all ESTs with similarity to a given *Arabidopsis* gene can perform an on-the-fly BLAST search of the *Arabidopsis* gene sequence against the collection EST sequences. A simple "BLAST View" is displayed in response, showing the query sequence along the top, and the Hits to each subject sequence stacked below it (Fig 3.6). This is similar to the cartoon displayed with BLAST results from many current online BLAST search sites.

Brassica oligos

Oligonucleotide sequences (oligos) were designed from the AAFC collection of *Brassica napus* ESTs in order to design a *Brassica* specific microarray. Where the oligos were designed from contigs with similarity to the *Arabidopsis* genome, it was possible to represent the oligos within BioViz. While the EST assembly had similarity to the *Arabidopsis* gene, the oligo may have been designed in a region lacking similarity, in which case it was drawn offset from the assembly (Fig 3.7).

Expression data

While it is not available in the current version of the browser, the display of expression data in the genome browser was attempted for Serial Analysis of Gene Expression (SAGE) data [58]. A SAGE tag is a 10bp tag extracted from a specific position of an expressed mRNA – the canonical site is downstream of the $3'$-most *NlaIII* restriction site (nucleotide pattern "CATG"). For a given set of SAGE tags in a library a rectangle was drawn at the position (relative to the chromosome sequence) where the tag occurs in the genome. The height of the rectangle was set to reflect the abundance of the tag in the SAGE library. This representation was used to look for localized "islands" of expressed genes and can be thought of as a type of histogram along the length of the chromosome. However, no such islands were identified in the SAGE data for which this was

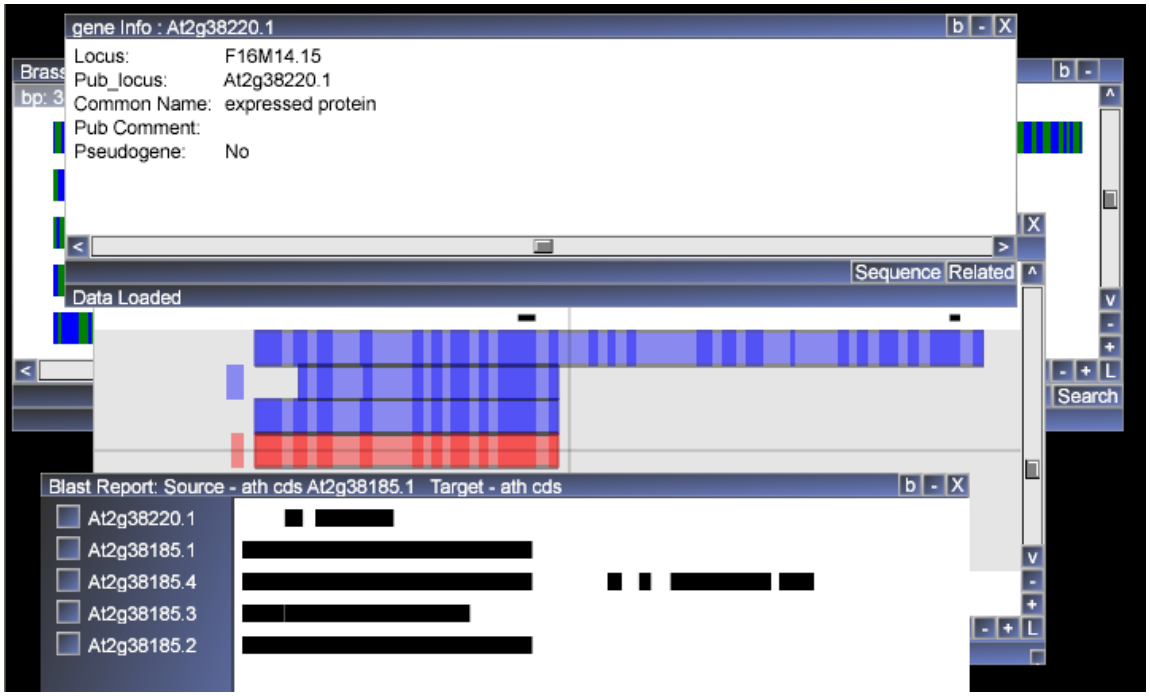

Figure 3.6: BioViz BLAST result display

Sequences having homology to the query sequence are represented within the width of the BLAST result view. Clicking on the name of the sequence takes the user to a detailed description of the sequence (shown for the homologous "At2g38220.1" sequence), while clicking on the rectangular HSP representation takes the user to the BLAST result details (not shown). This example shows all coding sequences with homology to the previously examined gene "At2g38185.1". Not surprisingly the 3 alternative splicing forms of the gene are identified (.2, .3, .4). More interesting is the homology with the second half of the ".4" splice form, which suggest a duplication event in this gene's history.

attempted and consequently the functionality was not maintained in BioViz. Details of the SAGE work have been published separately [49] and are not discussed in this thesis work because they are not related to the BioViz application.

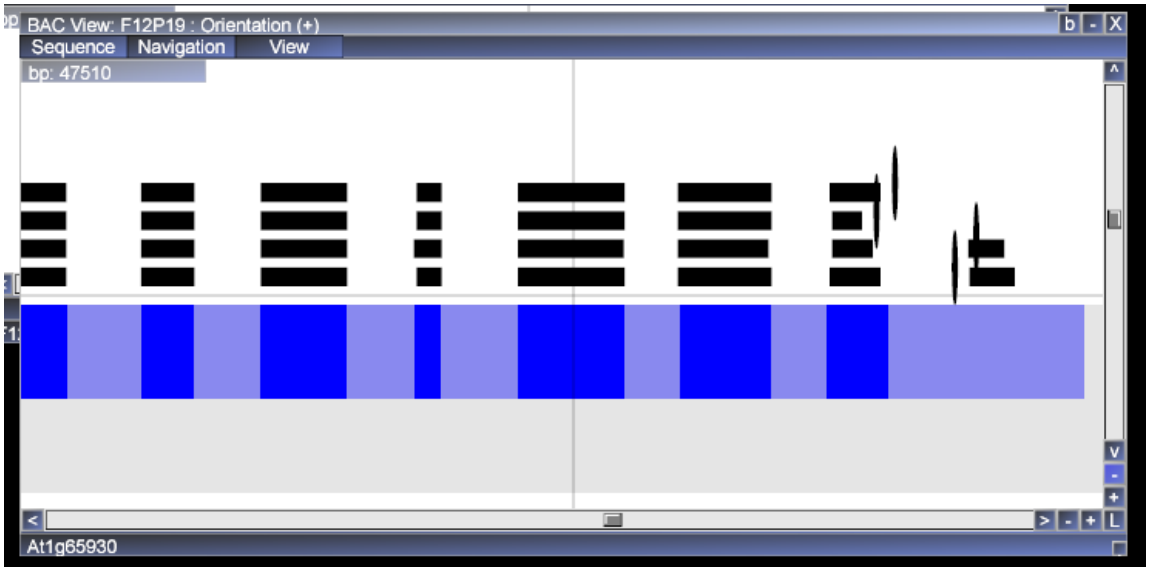

Figure 3.7: BioViz *Brassica* Oligo Display
Oligos designed for the AAFC *Brassica*-specific oligo array were displayed in the browser to provide users with access to the annotation of homologous *Arabidopsis* sequences. The *Brassica* assemblies (contigs) from which the oligos were designed were aligned with the BAC sequences, and the position of the oligo was marked with a circle (which appears as a narrow ellipse in the screenshot because it is compressed). Oligos which overlap the contig sequence (such as the one third from the top) are designed in a region with homology to *Arabidopsis* and as such might be expected to hybridize with *Arabidopsis*. However, the other three oligos are designed in regions without similarity to *Arabidopsis* and as such should be expected to be *Brassica* specific.

3.3.2 Performance characterization

It was suggested that the approach taken in Bioviz would reduce the number of page reloads and in turn reduce the amount of data transferred to the client. To test this suggestion a simple scenario was performed in both BioViz and GBrowse while logging network traffic. Network data was recorded using Microsoft Network Monitor 3.2 on the client-side. Data captures were filtered to include only packets sent to or received from the server. A new capture was performed for each of the steps described below. A summary of the data transfered for each step is summarized in Table 3.1. The number of data requests as well as the time spent and data transferred per request is summarized in Table 3.2.

Step 1, "Initial Load" involved recording the data transferred while the application loaded. In the case of GBrowse, it was necessary to delete cookies and clear the cache to ensure that all scripts

and icons were reloaded. Step 2, "Load BAC" consisted of loading BAC F16M14. In BioViz, this was accomplished using the "Find Sequence" option available from the Search menu on the initial Chromosome View. In GBrowse, this was accomplished by searching for the BAC in the search box. Step 3, "Load AAFC EST Contigs" involved adding the AAFC EST Contig representations to the currently displayed BAC image. In BioViz, this involves requesting supplementary information which is incorporated into the existing image. In GBrowse, this involves requesting a new image from the server. Step 4, "Zoom to Gene" involved zooming to gene At2g38180 using the browser navigation controls. In BioViz, this is performed entirely on the client-side, with the transformation being adjusted as necessary to focus on the gene and reveal its structure. In GBrowse, this involved transferring a new image from the server for each adjustment. Step 4, "Examine Neighbours" involved navigating to the two genes flanking At2g38180. Again, in BioViz this was accomplished entirely on the client-side, while GBrowse loaded new data for each transformation required to display the neighbours.

For the test, the GBrowse image width was set to 1024px, while no such setting applies to BioViz as the user can scale the image to their preferred size on the client-side. The following GBrowse tracks were enabled to make the displayed information approximately equivalent to that of the BioViz BAC View: BACs, Locus, and Protein Coding Gene Models. The representation for the Locus and Protein Coding Gene Models were set to "compact" to hide the descriptions, which are not available in the BioViz BAC View. Both BioViz and GBrowse were running on the same internal AAFC server, with tests performed in parallel to minimize any load differences on the server.

3.4 Discussion and Conclusions

3.4.1 Comparison with representations used in other browsers

The data representations used in BioViz differ from those used in other web-based genome browsers. This is partially a result of the "comparative genomics" emphasis within BioViz. For instance, all of the features which are part of the model organism (*Arabidopsis*) are contained within the bounds of the "chromosome" rectangle and other features are outside these bounds (Fig 3.4). This separates the two organisms, whereas the other comparable web-based browsers treat all features the same, so gene features and homologous regions from other organisms appear together in tracks below the genome (Fig 3.1). There is still a type of separation between the model organism and other organisms, but it is not so clearly defined.

BioViz displays features above and below the centerline to represent the orientation of the features (both genome features such as genes, and homologous sequences). This is compared to the directed arrow used to display features in the other comparable genome browsers (Fig 3.8). The fact

Table 3.1: Data transfer (bytes) in BioViz and GBrowse

		Support Files	Content	Total
BioViz	Initial Load	121,696	206,513	328,209
	Load BAC	-	21,047	21,047
	Load AAFC EST Contigs	-	17,595	17,595
	Zoom To Gene	-	-	-
	Examine Neighbours	-	-	-
	Total	121,696	245,155	**366,851**
Gbrowse	Initial Load	345,296	68,750	414,046
	Load BAC	-	95,813	95,813
	Load AAFC EST Contigs	-	118,028	118,028
	Zoom To Gene	-	3,417,252	3,417,252
	Examine Neighbours	-	277,506	277,506
	Total	345,296	3,977,349	**4,322,645**

Network traffic was logged while executing a 5 step use case in both BioViz and GBrowse. "Initial load" reports the data transferred when starting the genome browser. "Load BAC" reports the additional data transferred to display BAC F16M14. "Load AAFC EST Contigs" reports the data transferred to add the AAFC EST Contigs on BAC F16M14 to the representation. "Zoom to Gene" reports the data used to zoom into gene At2g38180 using the browsers navigation controls. "Examine neighbours" reports the data transfered while navigating to the two genes flanking At2g38180. The data transfer recorded in bytes and broken down into two parts, support files (e.g. style sheets and scripts) and content (e.g. the graphic representations, including the user interface elements in the case of BioViz).

Table 3.2: Time (seconds) and data (bytes) per request in BioViz and GBrowse

		Requests	Data per Request	Time	Time per Request
BioViz	Initial Load	1	328,209	3.89	3.89
	Load BAC	1	21,047	0.48	0.48
	Load AAFC EST Contigs	1	17,595	0.47	0.47
	Zoom To Gene	0	-	-	-
	Examine Neighbours	0	-	-	-
	Total or ***Average***	**3**	***73,37***	**4.84**	***0.97***
Gbrowse	Initial Load	1	414,046	14.36	14.36
	Load BAC	1	95,813	4.27	4.27
	Load AAFC EST Contigs	1	118,028	4.78	4.78
	Zoom To Gene	31	110,234	327.52	10.57
	Examine Neighbours	3	92,502	37.03	12.34
	Total or ***Average***	**37**	***166,125***	**388.00**	***9.26***

In BioViz, the test scenario involved one page load on startup and two additional data requests for additional data, after which all transformations occurred on the client side. In GBrowse, the test scenario triggered a page load on startup and one page reload each time additional data was requested or a new view was required. BioViz requested data from the server 3 times, with an average response size of 73Kb and an average response time of 0.97 seconds. Time per request is not reported for the transformation related steps (4 and 5) in BioViz as no data was requested from the server. GBrowse requested data from the server 37 times with an average response size of 166Kb and an average response time (including user time) of 9.26 seconds.

that the orientation of features differs is easier to observe at high-levels of zoom in BioViz. Even when zoomed in, it can be difficult to distinguish the orientation of some sequences in the other browsers. However, in BioViz it can be confusing to interpret the orientation of the feature until one is familiar with BioViz. Adding arrow heads to the features in BioViz would be an improvement to the existing representation.

It is straight-forwrad to identify sequences resulting from reverse transcription in BioViz. Such sequences appear on the opposite side of the centerline from the gene representation (Fig 3.4 d). However, in the other browsers genes on the chromosome are all displayed in the same track, with only an arrow to indicate orientation. Thus it is necessary to compare arrow heads between features on different tracks to identify potential reverse transcription sequences. Similarly, an EST having similarity to a region without an annotated gene sequence would more clearly standout in the BioViz representation because the EST would be aligned with an empty region of the genome. Whereas in the other representations, one would have to look at two different tracks for overlapping features and the user might overlook it in the array of available tracks.

3.4.2 Advantages of BioViz

The web-based genome browsers that were developed around the same time as BioViz exhibited two technology-imposed characteristics that BioViz sought to avoid: frequent page reloads and/or static layouts. BioViz avoids both of these characteristics through the combined use of SVG 2.4.1 and an AJAX-like approach (c.f. 2.4.2). While page reloads and a static display do not impede the creation of a functional browser, it was felt that they diminished the user experience. The scenario described in the previous section required 37 page reloads in GBrowse, with an average time requirement of 9.26 seconds per reload (Table 3.2). On the other hand, BioViz required only a single page load on startup, and required less than 1 second to retrieve additional data from the server 3.2).

Responsiveness

The time required for the data transformation steps (Zoom To Gene, Examine Neighbours) in BioViz is not recorded in Table 3.2 because there was no data transfer involved. However, the author is able to perform the transformation steps in under 20 seconds (wall clock time). This is the time required to perform the transformations. That the user is able to perform a series of operations in BioViz in 20 seconds that required 354 seconds in GBrowse supports the notion that the approach used in BioViz improves the user experience. However, there are a number of factors that could contribute to the increased time in GBrowse.

Looking at the time required in GBrowse for the transformation steps it is possible to roughly estimate the time spent reloading data compared to the client-side time for each request (Server vs

Figure 3.8: Placement of features
Screenshots showing features in BioVoz (top) and GBrowse (bottom). In BioViz, features are placed above the center-line if they are in the same orientation as the BAC sequence and below the center-line if the are in the opposite orientation. In the other genome browsers the orientation is indicated by an arrow on the features.

Table 3.3: Server vs. Client time for Data Transformation Steps in BioViz and GBrowse

		Time	Server	Client
BioViz	Zoom To Gene	15	-	15
	Examine Neighbours	5	-	5
	Total	**20**	**-**	**20**
Gbrowse	Zoom To Gene	327.52	148.18	179.34
	Examine Neighbours	37.03	14.34	22.69
	Total	**354.55**	**162.52**	**202.03**

An estimate of the user time for the transformation steps in GBrowse can be made by subtracting the time to load the AAFC EST Contigs view in GBrowse from the time for each request in the transformation steps. Based on this estimate more than half the time required for the transformation steps was user time.

Client in Table 3.3) by subtracting the time to load the AAFC EST Contigs image (4.78 seconds) from the time per request. This estimate provides a lower bound of the client-side time given that the time to load each image decreases as the user zooms in (a result of the decreased image complexity). Based on these numbers, using client-side transformations would save approximately 162 seconds of Server time related to generating and reloading the images.

The estimated client-side time in GBrowse is considerably higher than the required in BioViz. The client-side time in GBrowse is spent on two tasks, user orientation and triggering the transformation. The orientation time is required because the page reload means that the user might need to scroll the page to re-display the browser image, and the gross transformations applied in GBrowse require the user to re-locate the region of interest and decide which transformation to apply next to zoom in on the desired gene. The act of triggering the transformation is negligible; in our experience it is the reorientation that is the time consuming factor. The same actions in BioViz require approximately 20 seconds, which is almost entirely time spent performing the transformations on the client side, there is no need to reorient between renderings because transformations are seamless and the adjustments are small. This suggests that a more fluid interface could save as much as 200 seconds in GBrowse.

Together, the server-side rendering and resulting page reloads cost the user more than 5 minutes (355 seconds to perform the actions compared with the 20 seconds in BioViz). This time is attributed to the page reloads and reorientation required after each page reload. We observe that a significant savings is produced by using client-side transformation and dynamically updating the document, and thus conclude that the client-side transformations used in BioViz do in fact provide an improved the user experience by significantly improving responsiveness of the application.

Table 3.4: Comparison of image size for common image formats

Graphic	SVG	SVGZ	GIF	JPEG	PNG
shapes	2 KB	.7 KB	26 KB	24 KB	21 KB
gradients	2 KB	.6 KB	62 KB	35 KB	41 KB
tiger	96 KB	31 KB	52 KB	56 KB	111 KB

The above table was originally prepared by Ken Sall (1999). The raster graphics were created from the SVG image using Paint Shop Pro. The original table has been updated to include the compressed SVG image (SVGZ). The table and associated images are available at `http://www.wdvl.com/Authoring/Languages/XML/SVG/DoingIt/size.html`. As noted by the author SVG is quite amenable to compression because it is a text based format.

Band width requirements

A potential secondary benefit of using SVG came in the form of lower bandwidth requirements because vector graphics are often smaller than the corresponding high-resolution raster graphic (Table 3.4 shows one of the original comparisons for SVG). Thus the use of a vector graphics format has the potential to save bandwidth, both when the image is initially transfered and by reducing the number of page reloads. As can be seen in Table 3.1 an order of magnitude less data was transferred during this scenario using BioViz than using GBrowse, and there were 34 fewer data requests in BioViz than in GBrowse.

The page reloads present in the existing browsers occurred each time the user changed the current view - by zooming in on the chromosome or moving along the chromosome (scaling or translating). The use of a vector graphics format such as SVG can significantly reduce the number of required page reloads by eliminating the need to retrieve new data from the server for such minor view changes. Where a user is working on a slow internet connection or the server is under heavy load, keeping the transformation on the client-side should reduce lag and decrease the load on the server thereby make the browser feel more responsive. Interestingly, the under-development GBrowse version 2.0 is said to be a complete rewrite, which includes the addition of "slave renderer support" to "distribute reading databases and rendering tracks across multiple processes and machines" and thereby greatly improve performance [22]. This suggests that server-load can be a problem in the bitmap-based browsers, so pushing some of the data transformations to the client and reducing the number of required refreshes would be of value. In the scenario described in the previous section, 34 of the 37 reloads (and re-renderings) could have been avoided had the transformations been performed on the client-side (Table 3.2).

A second undesirable aspect of the page reload is the unresponsive period while the new page is retrieved from the server. This limits the rate at which the user can work to the speed at which new data can be retrieve from the server. A further consequence of this is that navigation can be imprecise. Unless the zoom is quite high it is only possible to make gross changes, which makes

tasks such as centering the view on a specific feature and then zoom-in awkward. Indeed, GBrowse version 2.0 is also implementing a new AJAX user interface to "provide a smoother user experience" [22]. In BioViz this is avoided by the use of a vector graphics to allow client-side transformations, and through the use of an AJAX-like implementation where data is received asynchronously and added to the existing page as it arrives. This allowed the user to continue navigating (drilling-down to new regions of the genome, zooming in on features of interest, opening gene annotations) while data was loading or searches were being performed. In the scenario described in the previous section, it seems that approximately 3 minutes (202 seconds) of user time might be saved as a result of implementing an AJAX style interface.

SVG

All the data images in BioViz can be scaled and translated on the client side because SVG is a vector graphics format (recall 2.4.1). Consequently, there is no need for the data to be reloaded each time the user desires a different perspective on already loaded data. The ability to transform the data on the client side improves the flow of browsing by reducing the number of times the client must fetch data from the server. For instance, genes have a specific structure that can only be visualized at high levels of magnification. With a bitmap-based browser, the client must wait for a new image from the server for each incremental change in magnification. Whereas BioViz allows the user to zoom in to the point where the structure is visible without loading any additional data.

Since the whole user interface is implemented in SVG, it is one large interactive graphic, which means that there are effectively no layout constraints imposed on the user. BioViz has been implemented using a "multi-windowed" format which provides the user with a great deal of flexibility when arranging the view of their data. The browser acts as a type of "desktop" for the browser, allowing users to arrange their views however they'd like or even to minimize views that they are not currently using. Given this implementation, users can view information about as many regions of the genome as they want, as many gene sequences, and as many features as they want (though they are limited by the physical space constraints of their display). A multi-windowed format inside a "desktop" format was selected rather than using multiple browser windows because of restrictions on inter-browser communication when working inside a plugin (browsers often restrict the ability of code in plugins to effect changes outside the plugin for security purposes).

To reiterate, being an XML+SVG based genome browser gives BioViz a number of advantages as compared to the standard bitmap based browser:

- Because SVG is a vector graphics format one can zoom and pan an SVG image on the client-side without loss of fidelity. This reduces the number of page reloads required, reducing load on the server and improving the responsiveness of the client. This also means there is no need need to wait for the new image to be returned from the server each time one would like to

examine the data from a different perspective.

- Because SVG is dynamic and interactive, new data can be added to the existing image and the image can be changed in response to user events (i.e. clicking on a gene).
- Because SVG is an XML based graphics format, developing XML technologies such as Extensible Stylesheet Language Transformations (XSLT) [9] can be used to transform XML formatted data into SVG. For instance one might transform MAGE-ML [52] formatted microarray experiments directly into SVG images for display or visual inspection within the browser.

3.5 Future Work

Planned improvements to BioViz include allowing users to perform sequence based searches, enabling searching by key words (which would ideally involve incorporating the Gene Ontology (GO) controlled vocabulary [4]). The work to mark the centromere on the chromosomes should be completed. Arrows should be added to the features in BioViz to make their orientation explicit.

Chapter 4

BioViz: Implementation

4.1 Synopsis

This chapter describes the implementation of the BioViz genome browser, including examples of the use of SVG and ECMAScript objects on the client-side and a discussion of the server-side framework.[1]

4.2 Background

The SVG format (c.f. 2.4.1) is used to provide both the data representation and user interface (UI) in BioViz. However, user interface elements are not defined in the SVG specification, so an object-oriented (OO) JavaScript library was implemented to provide the UI elements. On the server-side, OO Perl (c.f. 2.4.2) was used to handle client requests and produce the data representations. This background section shows an example of the use of both OO JavaScript and Perl.

4.2.1 Object-oriented JavaScript

ECMAScript (the standard upon which JavaScript is based) provides prototype-based objects rather than the more common class-based objects. The fundamental difference between the two types of objects being that in a prototype-based language there is no distinction between the "class" and an "instance" of the class. Instead, any object can act as a prototype for a new object that can be cloned to create a copy of the object with predefined initial properties. Any function in JavaScript can construct an object if called with the "new" operator. It is possible to implement inheritance within this paradigm by creating two constructor methods and assigning the constructor for the superclass as the prototype for the sub-class. This section shows how inheritance was implemented in the CGUI library using examples from the implementation.

[1] Portions of the material in this chapter were included in the early papers: "BioViz: Genome Viewer, Development of an SVG GUI for the visualization of genome data" [34] and "The Brassica / Arabidopsis Comparative Genome Browser, A novel approach to genome browsing" [36]; however, this chapter adds additional material related to the browser and the techniques used to generate the data representations used in the browser. CGUI has since been moved to a project on the open source software development site sourceforge and renamed CSVGUI (because there was already a CGUI library on sourceforge)

```
function CGUI (x, y, width, height) {
    this.width = width;
    this.height = height;
    this.x = x;
    this.y = y;
    return this;
}
```

Figure 4.1: Definition of the CGUI object constructor
The CGUI constructor requires x, y, width and height values to initialize the corresponding object properties.

```
var widget = new CGUI (10, 20, 80, 20);
```

Figure 4.2: Creation of a CGUI object
This will create a generic 80x20 CGUI object at position 10,20. However, the visual representation of the CGUI widgets is defined by the CGUI subclasses so the above is artificial.

The CGUI object defines an “x”, “y”, “width”, “height” attribute (Fig 4.1). The CGUI object is the base object in the CGUI hierarchy (Fig 4.9). A CGUI object could be created and and initialized with “x” having a value of “10”, “y” having a value of “20”, “width” having a value of “80” and “height” having a value of “20” as shown in Figure 4.2. This object is extended to provide other UI elements, for instance a “Button” constructor is defined and the object made to extend the “CGUI” prototype as shown in Figure 4.3. The “Button” adds a “text” attribute and an “event” attribute to the CGUI “x”, “y”, “width”, and “height” attributes. The “text” attribute is to hold text to display on the button and the “event” attribute is to allow the programmer to specify a handler for user click events. Note that the “CGUI” object’s attributes are not re-declared in the “Button” prototype because they are inherited from the parent prototype.

There are two key aspects to the inheritance. The first is setting the “prototype” property for the subclass to ensure that the class inherits the parent’s attributes and values. By setting the “prototype” property of “Button” equal to the CGUI constructor, we give the “Button” class all of the “CGUI” classes attributes (“x”, “y”, “width”, “height”) in addition to the attributes declared for the Button (“text”, and “event”). The second is initializing the inherited attributes. In CGUI, this was done by initializing a “base” attribute to reference the superclass constructor (“this.base = CGUI”), and then calling the constructor with the appropriate parameters.

An alternative to using the base attribute would be to use the JavaScript “call” method, which allows you to call a method on an object other than itself. This has the effect of making the “this” attribute in the method refer to the specified object instance. So the lines related to the base attribute could be replaced by a single “call” line as in Figure 4.4, which will have the effect of having the CGUI function update the attributes of the “Button” object.

```
function Button (width, height, text, event) {
    this.text = text;
    this.base = CGUI;
    this.base (0, 0, width, height);
    this.event = event;
    return this;
}
Button.prototype = new CGUI;
```

Figure 4.3: Definition of the Button object constructor and inheritance in JavaScript

In CGUI, the Button is an example of a CGUI widget for which there is a visual representation. The button constructor requires width, height, text, and event values to initialize the widget. The width and height attributes are inherited from the CGUI object, and as such are not initialized here. Instead, the CGUI constructor is called (this.base) with the width and height values. This is discussed further in the text. The text value is used as a label for the button, while the event value is a reference to an event handler to be called when the button is selected.

```
function Button (width, height, text, event) {
    this.text = text;
    CGUI.call (this, 0, 0, width, height);
    this.event = event;
    return this;
}
Button.prototype = new CGUI;
```

Figure 4.4: Alternative definition of the Button object constructor using JavaScript "call" method

As discussed in the text, the call method provides an alternative technique for initializing the superclass.

```
package Sequence;

sub new {
    my ($class, %args) = @_;
    my $self = {};
    bless $self, $class;
    if ($args->{'-id'}) {
        $self->{'id'} = $args->{'-id'};
    }
    if ($args->{'-sequence'}) {
        $self->{'sequence'} = $args->{'-sequence'};
    }
    if ($args->{'-alphabet'}) {
        $self->{'alphabet'} = $args->{'-alphabet'};
    }
    return $self;
}

sub draw {
    # implementation to be described later
}
```

Figure 4.5: Declaration of Sequence class in Perl.
The above Perl fragment defines a "Sequence" class by declaring a constructor ("new") in the "Sequence" package. The constructor allows the "id", "sequence", and "alphabet" of any newly created "Sequence" to be specified. It also defines a "draw" method (to be discussed later), which can be used to return an SVG fragment representing this "Sequence".

4.2.2 Object-oriented Perl

As compared to ECMAScript, Perl uses the more usual class-based objects. When working with Class-based objects, one pre-declares the methods and attributes of the "Class", which then serves as a template for the creation of "instances" of the object (though strictly speaking, in Perl you don't need to declare attributes). Changes to an "instance" have no effect on the creation of subsequent "instances" which are created from the "Class" definition. This section provides an example of inheritance in Perl using examples from the BioViz server-side implementation (described in detail in section 4.3.3).

Two of the objects rendered in BioViz are "Sequences" and "Contigs". A "Sequence" is a named string of amino or nucleic acid characters and a "Contig" is a set of "Sequences" which have been assembled into a single consensus sequence. In Perl, a "Sequence" might be described as in Figure 4.5, which allows one to create an SVG representation of a DNA sequence as in Figure 4.6. A "Contig" can be modeled as a sub-class of "Sequence", which gives it all the necessary "Sequence" characteristics (id, sequence, alphabet) and allows it to be used in the same way as a regular "Sequence". To store the additional information related to the sequences used to form the

consensus sequence an indexed list of contig "members" is added (Fig 4.7).

```
my $sequence = new Sequence(
                -id=>"test1",
                -sequence=>"ATCCGATCCATCAGCT",
                -type=>"DNA"
            );
my $svg = $sequence->draw();
```

Figure 4.6: Creation of a Sequence object
The above code fragment creates a short DNA sequence named "test1" and creates the corresponding SVG fragment using the draw method.

```
package Contig;

    our @ISA = qw(Sequence);

    sub new {
        my ($class,%ARG) = @_;
        my $self = $class->SUPER::new(%ARG);
        bless ($self, $class);
        $self->{'members'} = {};
        return $self;
    }

    sub draw {
        # implementation to be described later
    }
```

Figure 4.7: Definition of a Contig in Perl
The BioViz Contig object consists of a set (a hash) of "member" sequences.

Inheritance in Perl is accomplished through the use of the "ISA" array, which contains the list of an object's parent objects. In BioViz, the "Contig" is-a child of "Sequence" (Fig 4.7). Initialization of the inherited attributes is accomplished in Perl by accessing the "SUPER" class (i.e. the parent) through the "class" attribute and calling its constructor (the "new" method).

4.3 Implementation

The browser was implemented using a client-server architecture where the client-side is responsible for data presentation and the server-side is responsible for data retrieval in response to client requests (Fig 4.8). The Adobe SVG Viewer provides a non-standard addition to the SVG specification in the form of the getURL and postURL methods, which allows a message to be sent to the server and a response returned via a registered callback function. Together with the parseXML method, which takes an XML fragment and turns it into an SVG Document Element, it is possible

to asynchronously request data from the server and incorporate it into the current image. This allows the user to request supplementary information and continue browsing while the data loads. API enhancements added in the SVG 1.2 specification make it possible to provide functionality similar to that available via the getURL and postURL methods in a standards compliant fashion.

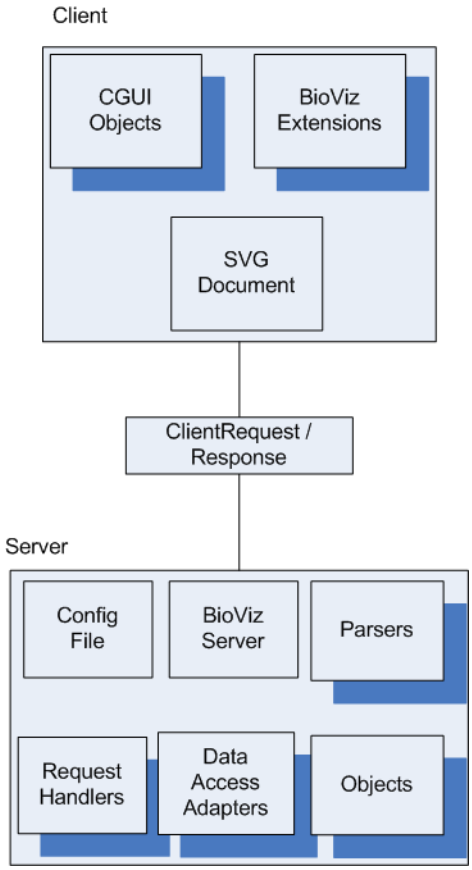

Figure 4.8: BioViz Architecture
BioViz consists of a Client-Server architecture with request-based communication between the two. The Client side is driven by an SVG Document, made up primarily of the CGUI widgets. The Server-side consists of a framework for handling client requests and generating the SVG Document fragments to be returned in response.

4.3.1 Client-side

The client of BioViz makes use of SVG in two ways. First, it uses SVG to display the content, and second, it creates user interface (UI) elements in SVG using JavaScript objects to provide interactivity. Content is returned from the server as SVG document fragments that can be inserted into the current display using the parseXML method and the Document Object Model (DOM) API. The UI elements are created using a hierarchy of objects developed to facilitate the creation of the UI (Fig 4.9). Each UI element is defined by a JavaScript object that can create a visual

representation of itself for insertion into the SVG Document. To facilitate the creation of BioViz specific views, objects were defined and included in the BioViz package.

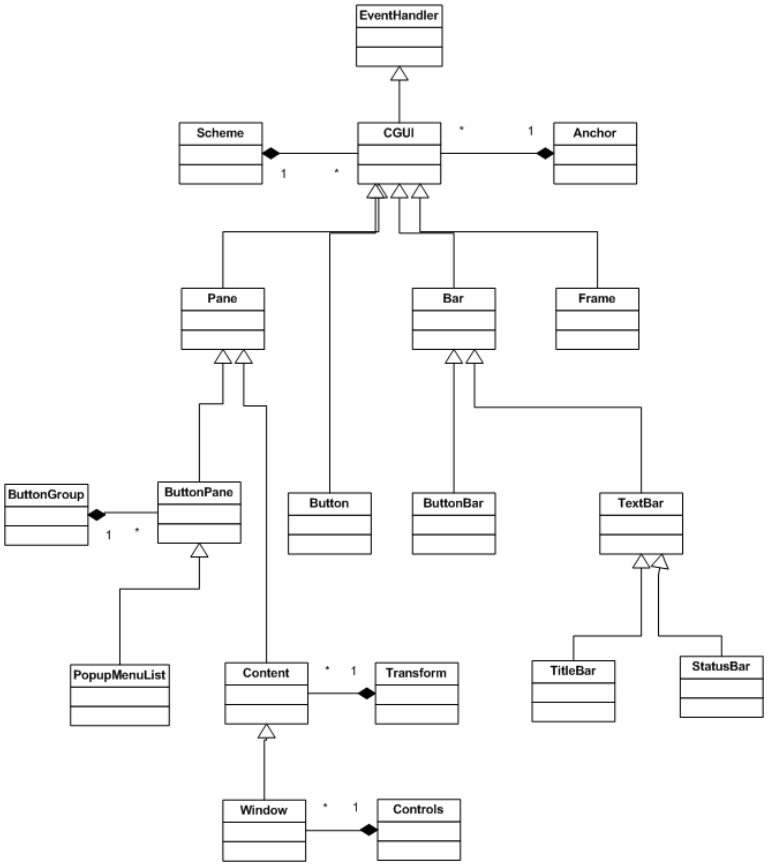

Figure 4.9: BioViz Client-side Architecture
The BioViz client-side architecture consists of an SVGDocument to define the web-applications view, the CGUI library to provide the UI elements used in the app, and bioviz specific CGUI subclasses to facilitate the creation of the various views used in BioViz. All CGUI UI elements extend the CGUI widget. There are 4 principal component types, three primitive: Pane, Bar, Button, and one composite: Frame. The composite Frame is composed from a number of other elements, one or more Bars and a Window. Via the root CGUI element all components inherit an Anchor object, which allows the UI element to be anchored at a specific location relative to another element. They also inherit a Scheme implementation which defines the visual appearance of the element. Note that some details have been excluded from the above diagram for the sake of clarity.

The key aspects of the client-side of the browser will be discussed here and illustrated using examples from the BioViz application.

"onload" event handler

When an SVGDocument is first loaded the "onload" event is fired [47]. Any new application being developed using CGUI needs to define an event handler and register it to handle the "onload"

event. In order to initialize the application, three things need to happen. The CGUI "init" method must be called to initialize the library. The postURLPath attribute of the CGUI object must be initialized with the path to which client requests should be submitted. The initial GUI for the application should be created; In BioViz this is accomplished using the "openMainFrame" method (Fig 4.10).

```
// called when the page is first opened
function _onload (evt)
{
    CGUI.init();
    CGUI.postURLPath = "/cgi-bin/bioviz";
    openMainFrame();
}
```

Figure 4.10: BioViz "onload" event handler
The onload event hander needs to be defined in any app using CGUI. The CGUI init method initializes the CGUI widgets while the postURLPath contains the URL to which requests should be sent. The openMainFrame method is unique to BioViz; it is responsible for opening the initial "Chromosome" view.

The "openMainFrame" method creates the initial view displayed when starting BioViz (Fig 3.3). It does this by creating a CHRView object (Chromosome View — the parameters are x, y, width, height, title), which is a subclass of the generic CGUI Frame (which provides a window inside the browser). The CHRView has two attributes "sourceOrg" and "sourceSource", which determine the dataset to use when creating the chromosome representations. In Figure 4.11, the values are set to "ath" (for Arabidopsis thaliana) and "tigr" (The Institute for Genome Research) because this was the main source for the *Arabidopsis thaliana* genome information displayed in the browser. Providing different values could create a request for information related to other organisms, thereby allowing other organisms to be used in the browser. After creating the Frame it must be added to the display — in Figure 4.11 the addToParent method is used to add the frame to the root of the SVGDocument (referenced by CGUI.root for convenience), but it could also be used to add it to another Frame or some other CGUI component.

After creating the main frame, the five *Arabidopsis* chromosomes are loaded. This is accomplished using the "getChromosome" method of the "CHRView" object. This method constructs a request for a specific chromosome representation, sends it to the server, and adds the results to the display area of the "CHRView". The format of the requests used in BioViz is discussed in detail in Section 4.3.2.

The CGUI library is included at the end of the SVG Document using an xlink. XML Linking Language (XLink) [12] is a specification that allows elements to be inserted into XML documents. It is analogous to the hlink from HTML.

```
function openMainFrame () {

  // create main frame
  var main_frame = new CHRView (
          0, 0, 700, 235,
          "Brassica / Arabidopsis Comparative Genome Browser (v0.5)"
       );

  // specify the organism being displayed - ath = arabidopsis thaliana
  main_frame.sourceOrg ("ath");
  main_frame.sourceSource ("tigr");

  // add the frame to the svg document
  main_frame.addToParent (CGUI.root);

  // load chromosomes and specify y offset
  main_frame.getChromosome (1, 20);
  main_frame.getChromosome (2, 50);
  main_frame.getChromosome (3, 80);
  main_frame.getChromosome (4, 110);
  main_frame.getChromosome (5, 140);
}
```

Figure 4.11: BioViz openMainFrame method
The openMainFrame method creates the "Chromosome View" (CHRView), which is a subclass of the CGUI Frame that has been extended to provide useful BioViz specific attributes such as the "sourceOrg" and "sourceSource" (source organism and data source), which are used in the client requests sent to the server. It then requests the representation of the 5 *Arabidopsis* chromosomes.

```
<script type="text/JavaScript"
        xlink:href="./cgui_lib.js.gz"/>
```

Figure 4.12: Including the CGUI source
The CGUI library source files have been concatenated into one file and compressed. This source library is included in the SVG Document (and made available to the application) by way of an xlink (a mechanism for including an external file in an XML document).

4.3.2 Messages

Client requests are made using the Adobe PostURL method. This requires code on the client-side to generate the request and handle the response, and code on the server-side to receive the requests and generate the response. On the client-side, the parameters for the request are encoded in an HTTP POST message. Figure 4.13 shows the code used to retrieve the chromosome representations on startup. On the server-side this request is received and the "parameter=value" pairs are parsed and used to create a ClientRequest object, as defined in Figure 4.14. In the case of a "GetChromosome" request, there is no "comparative" element to the request, and the "ClientRequest" object that is created would not have a defined "target" attribute. The response to a message in BioViz is always either an XML fragment that can be added to the current document or an error message to be displayed to the user.

```
CHRView.prototype.getChromosome = function (num, y) {

   window.postURL (CGUI.postURLPath + "BioVizServer.pl",
                   "service=GetChromosome" +
                   "&" + "source_id=" + num +
                   "&" + "source_organism=" + this. sourceOrg +
                   "&" + "source_type=chromosome" +
                   "&" + "source_source=" + this.sourceSource +
                   "&" + "y_offset=" + y, this.handler);

}
```

Figure 4.13: BioViz GetChromosome Message
The request parameters "source_id", "source_organism", "source_type", "source_source" and "y_offset" are concatenated to form a string of "parameter=value" pairs, with each pair separated by a "&", and this string is passed to the server. For example, the complete request for *Arabidopsis* chromosome 1 would look like: "service=GetChromosome&source_id=1&source_organism=ath& source_type=chromosome&source_source=tigr&y_offset=20".

4.3.3 Server-side

The server-side of BioViz was initially developed as a set of Perl CGI's, one for each client request that could be generated by the client. These individual CGI's were later merged into a single Perl module, which was then further redeveloped into a framework for returning SVG document fragments in response to client requests (Fig 4.15). Additional client requests can be handled by defining a new request handler and registering it with the framework.

When a client request is received, the framework determines the appropriate request handler and forwards the ClientRequest object to the request handler. The request handler is responsible for gathering data from the appropriate source and creating an SVG document fragment to be returned

```
package bioviz::obj::messages::ClientRequest;

# ClientRequest constructor
sub new {

        ...

        # String: the requested service
        $self->{'service'} = "";

        # A bioviz::obj::Provenance object:
        # Identifies the source of the requested data
        $self->{'source'} = undef;

        # A bioviz::obj::Provenance object:
        # Identifies the target for the requested data in the case
        # of a comparative request (such as a BLAST result)
        $self->{'target'} = undef;

        ...
}

package bioviz::obj::Provenance;

# Provenance constructor
sub new {

        ...

        # String: the source of the requested data
        $self->{'source'} = "";

        # String: the organism from which data is requested
        $self->{'organism'} = "";

        # String:  the id of the requested data
        $self->{'id'} = "";

        # String: the type of the requested data
        $self->{'type'} = "";

        ...
}
```

Figure 4.14: BioViz Message Format
A BioViz request consists of a requested "service" and two Provenance objects, which indicate where the "source" and "target" data should come from. The Provenance object defines a data source (institution), an "organism", a data "type" and an "id" for the requested information.

to and displayed in the client. Possible sources include XML files, flat files, SQL databases, and dynamically generated reports. The SVG returned from the server is added to the GUI using the parseXML method available in the Adobe SVG plugin.

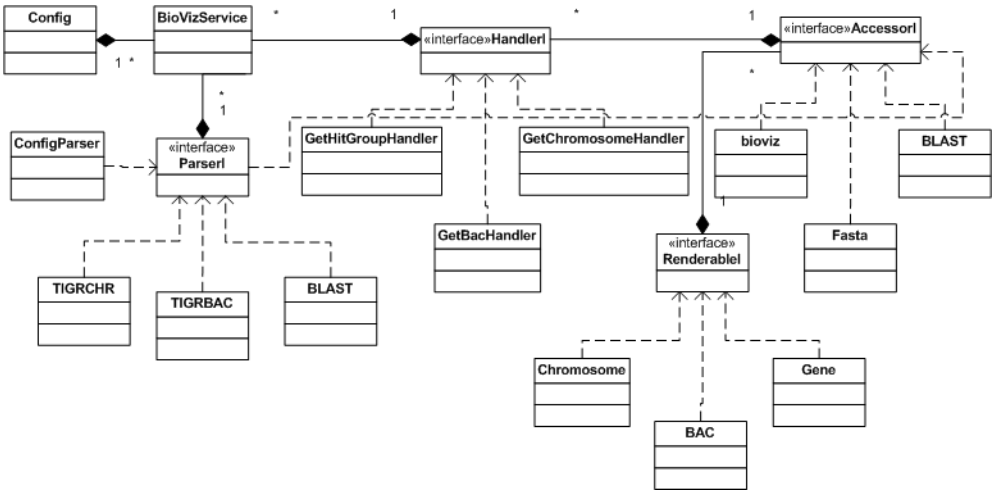

Figure 4.15: BioViz Server-side Architecture
The BioViz server-side architecture constists of four primary interfaces HandlerI, ParserI, AccessorI and RenderableI. Objects implementing ParserI are concerned with parsing text files, for instance BioViz requires an object to parse BLAST reports. Objects implementing AccessorI are responsible for accessing data, for instance, the "bioviz" accessor is used to access data in the MySQL database used in the AAFC BioViz install. Note that any object implementing ParserI is also an AccessorI. Objects implementing HandlerI are responsible for handling client requests, for instance, BioViz defines a "GetChromosomeHandler", which uses the "TIGRCHR" ParserI Object to parse and return the chromosome representation. Objects implementing RenderableI must be able to be rendered as SVG. These objects are used by the AccessorI objects to create the SVG content that is returned to the client. Note that some details have been excluded from the above diagram for the sake of clarity.

For example, in the case of the GetChromosome request described above, the ClientRequest is created from the parameters encoded in the HTTP POST submission. The ClientRequest is examined to determine the handler to use based on the specified "service" value. This results in the GetServiceHandler being selected. The GetServiceHandler is dynamically loaded, and the handleRequest method is called. This method uses the attributes of the "source" provinence object to determine which data source should be used to load the data by looking up the appropriate data source and accessor class in a configuration file. The use of a configuration file containing a datasource and accessor object ensures that different datasources can be configured without having to modify the GetChromosome request handler. This makes the GetChromosome request handler a type of framework class that can be made to return data for new organisms or from new data sources without modification.

4.3.4 AJAX

The browser has been implemented using a client-server architecture with the client-side responsible for presentation of the data, and the server-side responsible for retrieval of the data at the client's request. True to the AJAX approach (c.f. 2.4.2), supplementary information is asynchronously returned from the server and added to the existing SVG Document using JavaScript rather than having the page reload. Because the requests are asynchronous and there is no page reload the user can continue browsing while the data loads. Note that, while AJAX is now a common web development methodology, the term had yet to be coined at the time BioViz was developed.

4.4 Discussion and Conclusions

A version of the Viewer, with a subset of the functionality available in-house was made publicly available at www.brassica.ca [10] as a way of sharing the AAFC EST resource with the scientific community.

4.4.1 Client-side

Using an SVG based genome browser enabled interesting and creative techniques for viewing data relative to a model organism. The multi-windowed display and the asynchronous data retrieval present in the Brassica / Arabidopsis Comparative Genome Browser provided a more intuitive and productive browsing environment than standard bitmap based browsers. By applying an AJAX approach it would be possible to avoid page reloads in the bitmap based browsers. We are unaware of any existing web-based application that allows the same degree of flexibility in the user interface as was enabled through the use of SVG.

The ability to attach event handlers to specific SVG Elements makes it easy to create an interactive application. This applies equally to the UI Elements, where the event handlers were used to enable button clicks or frame movement, and to the data where it was possible to register an event handler on the data after it is loaded, for instance on the chromosomes to allow the user to drill-down to the BAC view. To do this using a bitmap based browser requires the generation of an image map for each image, which must be updated each time the image is adjusted. While this is entirely functional and feasible, it seems more cumbersome than attaching a listener directly to the target in the image.

The decision to use an SVG browser will restrict the audience to those able to display the content - though of late the majority of modern web browsers support SVG natively (that is, without a plugin). The need to reach the broadest possible audience must be balanced with the desire to do novel things. Ultimately the best decision might be to provide both interfaces because this

would allow the full spectrum of users access to the data, while providing a more flexible interface for those who are willing and able to use it. This could be accomplished using a common server-side implementation, which returns either SVG document fragments or complete bitmaps (to be integrated into the display using AJAX). Such an objective could, theoretically, be incrementally implemented for one of the more "full-featured" genome browsers by providing alternative SVG-based representations for some data over the course of time until a full parallel SVG implementation was available.

CGUI

The development of CGUI facilitated the creation of the interface for BioViz. Sub-classes of the core CGUI elements where defined with appropriate default values and application specific functionality. For instance, the views in BioViz were implemented as BioViz specific sub-classes of Frame (i.e. CHRView). The UI API defines methods to easily insert content returned from the server at the right place in the SVG Document to have it rendered, and to scale and translate the content to ensure the user can access the information they need.

The SVG user community was quite interested in the development of an SVG UI library, and there were a number of projects started to develop one. CGUI demonstrated that such a library was feasible and contributed to the larger community effort. To ensure that others could use the CGUI library, BioViz specific functionality was not included in the core CGUI library. Shortly after the initial release of BioViz, CGUI was released as an open source project to make it available to the community. As a result, a number of significant contributions were made to the toolkit in terms of additional widgets and the separation of the widget representation from the widget declaration, and in the end one of the major contributors took over maintenance of the version of the toolkit that was released on Sourceforge (a popular repository for open source projects). CGUI is available from Sourceforge (`http://csvgui.sourceforge.net/`) and 3rd parties built applications using the toolkit [46].

Limitations

Three key limitations related to the use of SVG in BioViz were reported in the earlier publications. Specifically, performance when rendering complicated images or large amounts of text, the lack of native SVG support in mainstream web-browsers and the resulting restrictions imposed by the use of a plugin. However, recent developments have removed these limitations.

Mainstream browsers (any WebKit based browser such as Safari, Chrome, or Konqueror; and Firefox) all natively support SVG which eliminates any restrictions related to the use of a plugin as well as eliminating the requirement for installing a plugin that might have prevented some users of the application.

A new SVG viewer, Renesis (`http://www.examotion.com/`), has been developed that is claimed to be much more efficient than the original Adobe SVG Viewer, and this should result in performance improvements in BioViz. If this new viewer significantly outperforms the native SVG implementations in the WebKit-based browsers and Firefox, it can only help but motivate further performance improvements in those implementations.

4.4.2 Server-side

The modular architecture employed in the redesigned BioViz server-side implementation is a marked improvement over the original "script-per-request" approach in that there is a generic high-level framework and a clearly defined interface for request handlers. This results in code that is easier to maintain and makes it easier to extend the application to handle new requests.

The framework allows new request types and request handlers to be defined in three easy steps. The new request type is added to the server-side configuration file, and mapped to a specific request handler, which uses a specific datasource to handle the request. The request handler and datasource are then implemented (assuming an existing datasource does not already provide the necessary data). This allows multiple datasources to be used in the same instance of the application, and would, for instance, allow BioViz to be configured to work from the database of one of the other genome browsers, for instance the GBrowse database. This is all accomplished without making any code changes within the framework or server-side scripts. The only coding necessary is to actually handle the request.

The request handlers are analogous to the original request handler scripts. However, the framework and defined interface enforce standards and a formal structure for the request handlers that contributes to a cleaner implementation and better organization. The end result is more readable and maintainable source code.

4.4.3 Installation

When installing the browser at a new location, the local database administrator (DBA) must create Perl modules to handle any client requests that are specific to their location, and data access modules to access their databases and the configuration file must then be updated accordingly on the server. This makes an out-of-the-box installation of the browser impossible — except in the unlikely event that the local site has the same table structure and same data as in the original install. However, it also allows the browser to access an existing database installation, and ensures that the browser is not tied to a particular database managements system or database schema. For instance, the browser could use a database as simple as a list of features and their regions of similarity relative to the *Arabidopsis* BACs or it could use an existing MySQL database installation, so long as the DBA provides an implementation to access the desired data source.

4.4.4 AJAX

As mentioned earlier, the GBrowse 2.0 implementation will be implemented with an AJAX style interface. However, an SVG-based browser such as BioViz might achieve an additional improvement by returning an SVG Document Fragment (a partial SVG Document) to be incorporated into the existing image. Whereas a bitmap based browser would need to return an entirely new image to be added to the containing HTML document (unless the graphic were made up from an array of smaller images). This should result in a savings in bandwidth and rendering time on the server as compared to sending a new image each time (recall section 3.4.2).

4.5 Future Work

BioViz, or more specifically CGUI, needs to be updated to function in the new Renesis SVG viewer, as well as the native web-browser SVG implementations. Renesis claims to be fully compatible with the latest Adobe SVG Viewer; however, unconfirmed reports by users of CGUI indicate that applications based on CGUI do not function with Renesis. Testing unrelated to BioViz found that SVG support between Opera and Firefox differs, so updating BioViz to function within the native browser support may prove complicated. However, the CGUI library has been updated by a thrid party to function in Firefox and Opera, which suggests that it should be possible.

Having updated BioViz to work with the new SVG implementations, it would be interesting to test the performance of BioViz on each of the different implementations. In the process of converting the application to work with the different implementations, a set of guidelines for ensuring cross-platform SVG support could be developed. There is already an official SVG test suite available, and the results of testing on the different SVG viewer implementations is reported. However, as with all web development, there are likely to be minor differences between the implementations due to varying interpretations of the specification and/or incomplete and/or incorrect implementations. For instance, in some recent, unrelated (unpublished) visualization work performed by the author, text rendering behaviour with respect to view ports was found to differ between Safari and Mozilla.

The client-side of BioViz, including the CGUI library could be reworked to take advantage of the functionality offered in open source libraries such as Prototype and JQuery for creating ECMAScript classes and interacting with the Document Object Model (DOM), which forms the basis of an SVG Document. This should simplify the BioViz and CGUI JavaScript considerably and capitalize on the community effort to create these robust, cross-browser compatible libraries. At the same time, it would be good to rework the JavaScript objects to make better use of the id attribute of the SVG Elements instead of maintaining references to the SVG Elements inside the JavaScript objects. The references to the SVG Elements were inserted as a result of what we feel to be a bug in the Adobe SVG Viewer, which allowed multiple elements to exist in the document with

the same id and resulted in erroneous behaviour when performing a lookup on the SVG Element.

The configuration mechanism in BioViz could be simplified. At the moment the config file is quite verbose because you must specify an entry for each valid request, including all valid parameter combinations (because the datasource or request handler to use might vary depending on the parameters). A more concise notation or a simple UI to configure the browser might be desirable.

Given that GBrowse 2.0 will be developed with an AJAX-style user interface, it would be interesting to characterize the performance difference between GBrowse 1.0/2.0 and BioViz. The difficulty would be in ensuring an "Apples-Apples" comparison. Measures that could be considered include time to render equivalent views between each of the implementations and bandwidth required for a series of "typical" operations. It is the author's expectation that the SVG-based implementation would prove more efficient than the bitmap-based implementation assuming all other variables were the same. This expectation is supported by the data in section 3.4.2. One way to do this would be to modify GBrowse to return SVG Format graphics instead of bitmaps. Another option would be to create standalone services that return graphics as bitmaps and SVG and then to compare the performance impact of zooming and panning as implemented in GBrowse. It would be especially important to try this on a slow network connection and a server that is under heavy load, because these are the two scenarios where the lower bandwidth and less frequent data requests of an SVG-based browser are expected to be most advantageous.

Chapter 5

BioViz: Impact

5.1 Synopsis

This chapter highlights the impact of BioViz outside the bioinformatics community, specifically the impact of the CGUI library development in the area of SVG UIs.[1]. This chapter proceeds as follows: first, necessary background regarding the CGUI and SPARK projects is presented, and then the steps taken to make the SPARK compliant CGUI (S-CGUI) widgets are outlined. Next the conversion guidelines and suggested extensions elucidated by this process are presented, and two sample applications created using the converted widgets are presented. The final pages summarize the results of this work and suggest topics for further investigation. Supplementary resources, including the widgets used in this paper and their source code are available at `http://homepage.usask.ca/~ctl271/cgui/spark`.

5.2 Background

The use of Extensible Markup Language (XML) (c.f. 2.4.1) for the definition of SVG images, coupled with the interactive nature of the resulting graphics resulted in considerable interest in the use of SVG for web application development. For the most part the developers of SVG enabled web applications have had to create their own widgets from scratch. This has resulted in many interesting variations of the same basic widgets at the cost of considerable duplicated effort.

In an attempt to make the creation of web applications easier there have been attempts to create standard SVG widget sets, among them the kevlindev.com widgets [37], the CGUI widgets [34, 30], the SVgUI project (a now defunct sourceforge project started by Kevin Lindsey, circa 2002) and the dSVG widgets from the Corel Smart Graphics Studio project (also defunct). However, even in these projects there was substantial duplication and a lack of interoperability between the resulting widgets.

[1] The paper from which much of the following chapter was derived [31] tested the general applicability of an SVG GUI framework [17] which was developed with input from the author. The paper was presented at the SVG Open 2005 conference in Enschede, Netherlands.

To promote interoperability and thereby decrease wasted effort, the originators of the SVG Programmer's Application Resource Kit (SPARK) project proposed an SVG GUI framework to provide guidelines for the construction of SVG widgets and enable interoperable widget sets. This project was founded by the thesis author in collaboration with Schemasoft, a software development company based in Vancover, BC (which has since been acquired by Apple). The first draft of the SPARK GUI Framework (SPARK-FW) was prepared by a student at Schemasoft with input from the thesis author and developers at Schemasoft [17]. However, only a limited number of widgets were created [33] and its general applicability was not, strictly speaking, demonstrated.

This project sought to evaluate the robustness of the SPARK framework by refactoring an existing, proven SVG GUI library (CGUI) [30] to make it SPARK compliant. CGUI was selected for this effort for three reasons, 1) the author's familiarity with the library, 2) that it was one of the only SVG GUI libraries still under active development, 3) that it had been proven effective through use in several third party applications. The process of updating the CGUI library to work within the SPARK-FW provided three key benefits: a set of guidelines that the developers of other SVG based widgets can employ when adapting their widgets to work within the SPARK-FW, a set of proposed extensions or modifications to the SPARK-FW based on needs identified by this more thorough application of the framework, and a demonstration that the proposed framework is generally applicable.

5.3 Implementations

5.3.1 CGUI

The CGUI widgets are implemented using ECMAScript objects to maximize code reuse and ensure the smallest possible size for the packaged source. The original widgets had the ability to render themselves using a hard coded graphical representation, though the visual representation was later separated from the widget and moved to external Schemes. The widgets also provide visual cues to indicate their state. For instance, buttons can be in the selected, active or disabled state and their appearance is altered accordingly. The downside of the CGUI widgets is that they are created imperatively, which doesn't appeal to some users who desire a declarative syntax more in keeping with the declarative nature of SVG. Conversion of the CGUI widgets to the SPARK-FW would provide this declarative syntax; however, it is important to maintain the original imperative API to support legacy applications.

The original CGUI design philosophy was that all widgets should inherit common functionality from the root CGUI class (Figure 5.1), that there would be a small number of basic widgets that could be extended to provide more sophisticated functionality (Figure 5.1) and that each widget would be responsible for generating its own view. There are three basic widgets, a Pane to contain

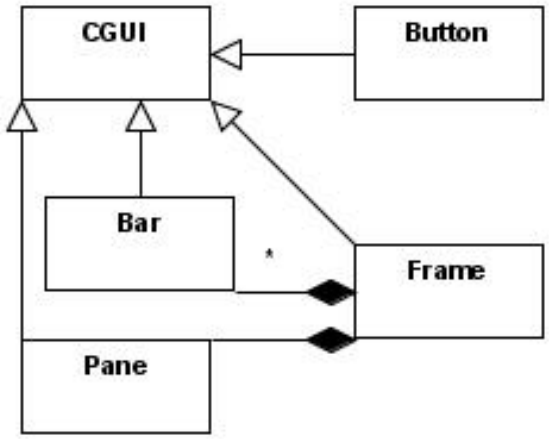

Figure 5.1: Root of the original CGUI hierarchy
There are three basic types Bar, Pane and Button and a composite type Frame which can be moved, resized, minimized and closed. The make up the root of the CGUI hierarchy (Figure 4.9).

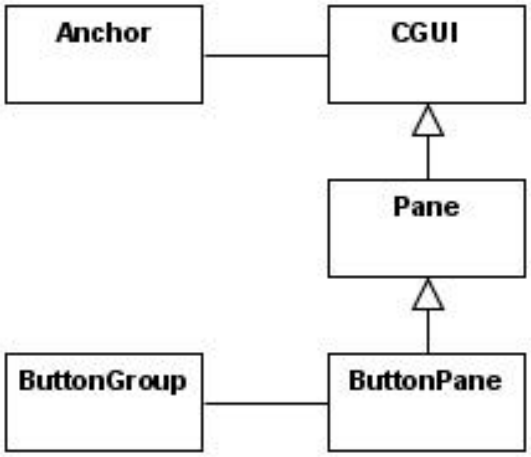

Figure 5.2: Example associate classes
Every widget has an anchor property inherited from CGUI that can be used to position the widget relative to some target SVG Element (it may be another widget or an Element in the SVG document). Similarly, every ButtonPane (a Pane containing a set of Buttons) has an associated ButtonGroup which controls the behavior of the set of buttons in the ButtonPane (for instance they might be set to behave as checkboxes or radiobuttons).

other widgets and act as a layout element (somewhat akin to a simple java.awt.Pane), a Bar to contain text and buttons (which in hindsight is superfluous and should be absorbed into the Pane), and a Button to receive user input. The fourth type is a Frame, this is a compound widget made up of 1 or more Panes and 1 or more Bars to allow them to be moved as a group, minimized, etc. thereby creating a Window (somewhat akin to the java.awt.Frame). Classes without a visual representation, for instance helper classes, do not inherit from the hierarchy but are instead associated with the appropriate widget(s) (Figure 5.2).

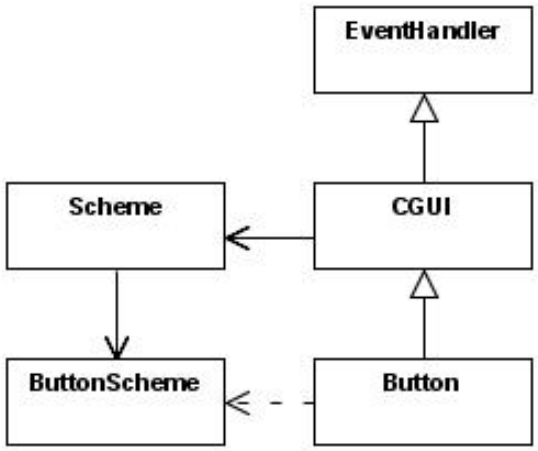

Figure 5.3: Root after architectural change
The above diagram shows the root of the CGUI hierarchy at the time of the SPARK work. Each widget, for instance the "Button" shown in the above extends the "CGUI" object, which in turn extends the "EventHandler". Each "CGUI" element has an associated "Scheme" object, a subtype of which definition each widget's representation.

There has been one architectural change to the CGUI library since the initial version, and it is this modified version that was used for the SPARK trial work. The change involved the separation of the widget's appearance from the widgets themselves. The definition of the widget's view was moved into a separate Scheme class, one for each widget, and a collection of Scheme objects is assigned by the programmer when the application is initialized. Each widget uses the corresponding scheme in the assigned collection unless the designer overrides the scheme associated with an individual widget or instance of a widget. At this time all widgets were also made to extend the EventHandler class, which provides common event registration/deregistration and event handling methods to all widgets (Figure 5.3). The intent here was to extend this class to provide a queue allowing multiple event handlers to be registered for the same event type on a given widget, though this was never implemented.

5.3.2 SPARK

The SPARK-FW defines a root hierarchy that is arguably similar to the original CGUI hierarchy (Figure 5.4). In the SPARK-FW each widget inherits common functionality from the root Widget class and there are a number of basic widgets that must be extended to provide more sophisticated functionality. These basic types closely resemble those of the CGUI hierarchy in that one provides a mechanism for receiving user input (Atom) and the other is used to group related widgets (Container).

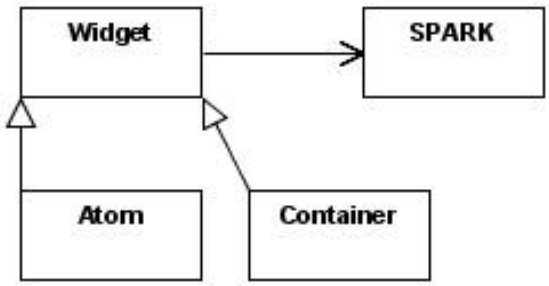

Figure 5.4: SPARK Root Hierarchy
The root class in the SPARK hierarchy is the Widget. Atom and Container are two abstract classes that extend that define basic widget types. Each widget in the SPARK framework can access core functionality available through the SPARK class.

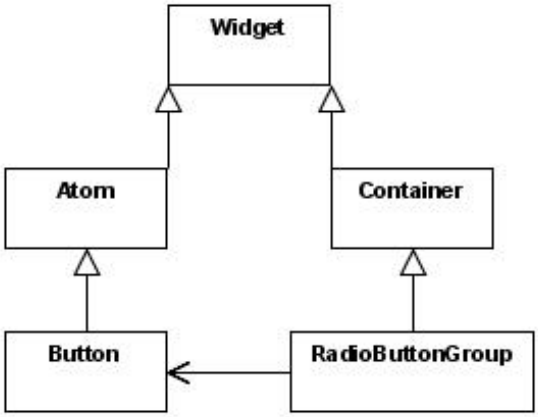

Figure 5.5: Implementation of RadioButtonGroup in SPARK
A RadioButtonGroup is a Container containing some number of buttons. The RadioButtonGroup contains the logic related to implementing the RadioButton behavior.

However, there are a number of philosophical differences between the SPARK widgets and the CGUI widgets. For instance, while the SPARK Container and CGUI Pane are conceptually similar

they come with different expectations. It is expected that a Pane will establish a viewBox and that items contained by a Pane but extending outside the visible area will not be displayed (though this expectation has relaxed over the course of time), whereas there is no such expectation of the SPARK Container. Another example can be seen on examination of the CGUI ButtonPane. In CGUI the RadioButton behavior is delegated to a non-widget so that ButtonGroup can be used without requiring that the Buttons appear in a particular Pane (Figure 5.2). Whereas in the SPARK examples, the RadioButtonGroup is a Container, so Buttons which are part of that group cannot be placed in other Containers (Figure 5.5). Fortunately, such philosophical differences need not interfere with the effort to make S-CGUI widgets as such decisions are not enforced by the SPARK framework and are left up to the designer.

```
1  - <g id="b2" transform="translate(300, 100)"
2  -    class="SPARK atom Button Style 1">

3  -    <desc>Button 2</desc>
4  -    <metadata>bar</metadata>

5  -    <g><!-- define view -->
6  -        <elipse rx="50" ry="25">
7  -            <animate attributeType="xml" attributeName="ry"
8  -                from="25"' to="35" dur="0.2s"
9  -                begin="b2.mouseover" fill="freeze"/>
10 -            <animate attributeType="xml" attributeName="ry"
11 -                from="35" to="25" dur="0.2s"
12 -                begin="b2.mouseout" fill="freeze"/>
13 -        </elipse>
14 -    </g>
15 - </g>
```

Figure 5.6: Creation of a SPARK Button
The above SVG Fragment shows the declarative creation of a SPARK Button. The lines have been prefixed with a line number for ease of explanation. The <g> (group) element (Lines 1-2) positions the button at x=300, y=100 relative to its parent element and declares the type in its class attribute; in this case the widget is a SPARK atom of type Button. The portion "Style 1" can be used by the applied Cascading Style Sheet (CSS) to style the button. The view of this widget (lines 5-16) is an ellipse that expands, via declarative animation, when the mouse is over the button and then returns to its base state when the mouse moves off the widget.

The declarative nature of the SPARK-FW necessitates two helper classes that were not necessary in CGUI. The first of these is the SPARKFactory. This class provides methods for creating SPARK compliant widgets. When writing new SPARK compliant widgets the designer would add these widgets to the Factory's list of known widgets, and the factory is responsible for creating the corresponding object when an element with the target class attribute is encountered in the SVG Document. The second helper class is the SPARKDecorator, which is responsible for adding the appropriate functionality to the widgets as they are created. The designer would typically write

Figure 5.7: Two SPARK buttons
The button on the right corresponds to the button declared in Figure 5.6 above. The screen shot was taken with the mouse over the button on the right, hence it is slightly larger than the button on the left. The two buttons have a different visual appearance as a CSS has been applied to the button on the right.

a custom decorator for their application to assign the appropriate functionality to the widgets. The designer need only modify the SPARKFactory if they are adding new widgets for use in the SPARKFramework.

5.3.3 SPARK & CGUI

SPARK compliance offers several advantages to users of the CGUI widget set. The primary advantage is that S-CGUI widgets can be created declaratively and should be interoperable with other SPARK compliant widgets as they become available. An additional benefit is the ease with which S-CGUI widgets could have their appearance (skin) changed. This is because the skin can be created declaratively at the same time the UI is described in the SVG document. Thus the designer can provide virtually any representation of the widgets at the time that the UI is created (Figures 5.6, 5.7), whereas a user of the CGUI widgets must code a custom skin. Because the UI is pure SVG it is amenable to creation using a standard SVG Editor, whereas a user must know JavaScript in order to create a CGUI skin.

Despite these advantages there are several important considerations when adapting the CGUI widgets. While a declarative syntax for creating the widgets is desirable, there may be circumstances where an imperative interface is preferable and/or necessary. Thus all efforts should be made to ensure that the widgets can be still be created imperatively after they become SPARK compliant. Furthermore, where possible the old imperative interface to the CGUI widgets should be preserved to minimize the impact of the changes on existing applications.

5.4 Questions

While planning the work required for the paper from which this chapter is derived, five questions were asked:

- What is required to make the widgets SPARK compliant?
- Can a converted CGUI widget be used within the SPARK framework?
- How is functionality added to the SPARK widgets?
- Can the widgets be declared more concisely?
- Are the converted widgets interoperable with the existing SPARK widgets?

The questions had to be answered in order because each question builds on the previous result. The primary objective in asking these questions was to determine whether or not it would be possible to make the CGUI widgets SPARK compliant, while the secondary objective was to explore any possible extensions to the SPARK-FW that arose in the course of the work. The answers to these questions are briefly described here.

5.4.1 Making the widgets SPARK compliant

In order for the widgets to function within the SPARK framework, they must extend the root of the SPARK hierarchy. If they do not, they will not have the appropriate interfaces and it will not be possible to create them using the framework. A second requirement is that the widgets must use the Command pattern [21], because this is the method by which functionality is attached to widgets in the SPARK framework. A third requirement is that it must be possible to declare the widgets declaratively — this last requirement was one of the main reasons for making the widgets SPARK compliant.

Extending the SPARK hierarchy

SPARK compliant widgets should inherit from the SPARK hierarchy. In general, this means extending one of either Atom or Container, whichever is the most appropriate for the new widget. As the CGUI widgets do not extend the SPARK hierarchy, this seems like a fairly obvious starting point in the conversion process. Unfortunately this process is complicated by the fact that all of the widgets already extend the CGUI hierarchy.

The straight-forward approach to having the CGUI widgets extend the SPARK framework is to have them inherit from both the CGUI and SPARK hierarchies. Of course there are many arguments for avoiding multiple inheritance, so the "replace delegation with inheritance" refactoring [19] is used to maintain access to the CGUI core functionality while allowing the widgets to extend the SPARK classes (Figure 5.8). Delegating the CGUI functionality required creating an instance of the CGUI object inside the CGUI widget and making all calls to the inherited CGUI functionality via the new reference. In the case where the CGUI method was overridden or extended, the widget's

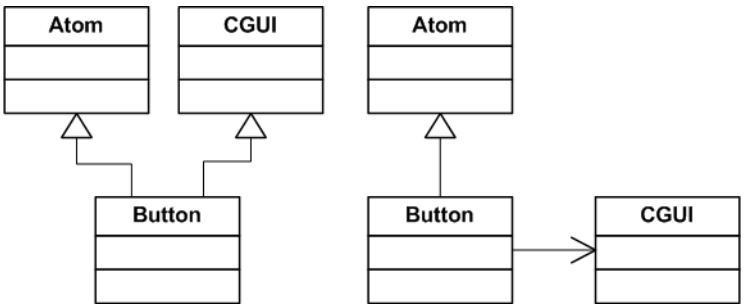

Figure 5.8: Replace inheritance with delegation
(Left) An example showing the CGUI Button inheriting from the CGUI and Atom objects. (Right) The refactored Button, which inherits from the Atom object and has a reference to the CGUI object now satisfies the first identified requirement for SPARK compliance.

method will now call the method on the CGUI instance and perform the necessary additional functionality as appropriate.

Allowing the CGUI widgets to use Commands

In the CGUI framework event handler code can be passed to widgets such as a Button or MenuItem, which is called when the widget handles the mousedown event. In the SPARK framework, this event handler code is provided in the form of Commands, and multiple commands can be added to a given widget. By extending the SPARK framework, the old CGUI widget inherits the addCommand and runCommands functionality. In CGUI the line of code that executes the event handler code can be replaced with a call to runCommands, and the code allowing an event handler to be assigned can be removed. After this the old EventHandlers can be trivially converted into Commands and added to the widget using the addCommand method.

Allow the widgets to be created declaratively

The SPARK-FW requires that widget constructors accept a single argument, the DOM tree describing the widget; this tree can be a subtree in a larger document or a document fragment. This requirement allows the widgets to be created by the SPARKFactory class in a single pass of the SVG document when the document is loaded and seems like a logical second in the conversion process. The challenge here is to change over the CGUI constructors, while preserving a simple imperative interface.

The SPARK constructor traverses the DOM subtree and saves references to nodes in the DOM as necessary. For instance, all SPARK widgets have an anchor property that references the root of the widget's subtree. This is a marked contrast to the CGUI widgets, which are created imperatively and take arguments as necessary to construct the object and create the view defined in the associated

Scheme.

There are really two problems here. First, the constructor needs to be modified so that it takes in the tree describing the widget as an argument. Second the logic which initializes the ECMAScript object must be preserved and the initialization parameters must be extracted from the DOM subtree. In order to preserve the initialization logic the "extract method" refactoring can be used to move all of this into an initialization method, and, at this time, the call to create the widgets view can be removed. Once this is complete, the arguments in the constructor can be replaced with the single input, and a parseDOM method added to extract the required initialization parameters and save references to nodes in the DOM. After the parameters have been extracted from the DOM, the object can be initialized using the old logic.

```
Old CGUI imperative declaration:

1 - var button = new Button(90, 20, "Button 1", controller);

Alternate hybrid imperative declaration:

1 - var node = Button.createScheme ("b3", 90, 20, "Button 1");
2 - parent.appendChild(node);
3 - var button = new Button (node);
4 - button.addCommand(new Message());
```

Figure 5.9: Imperative declaration of converted CGUI Button
The constructor for the old CGUI button took 4 arguments (as well as some optional arguments which have not been shown here). The width and height of the button, the label for the button and either an object implementing the EventListener interface or a method to handle the event. The new SPARK compliant Button takes in the root of the DOM tree representing the widget, and the width, height and label parameters from the old constructor have been moved to the createScheme method. The object implementing the EventListener interface has been replaced by a Command as per the SPARK framework.

This change allows the widget to be created declaratively using the SPARK-FW, but does not prevent the widget from being created imperatively. The tree describing the view can be constructed with the necessary parameters from the old constructor call, added to the SVG Document and then passed to the new constructor (Figure 5.9).

5.4.2 Using the new widgets within the SPARK framework

At this point, it appears that it should be possible to use the converted CGUI widgets in the SPARK-FW. The constructor has been modified to make it usable by the SPARKFactory, which should allow a UI to be declaratively created. All SPARK related functionality, for instance, the ability to add and execute Commands is available because the widgets extend the SPARK hierarchy. And finally, Commands added to the widgets will be executed.

Widgets in the SPARK-FW are created by the SPARKFactory when the SVG Document is loaded. Each node in the DOM is inspected, and if the class corresponds to a registered SPARK type, then the appropriate constructor is called via the Factory. In the original SPARKFactory there was a hard coded list of available widget types, and a hard coded constructor applied for each type. So to use a newly created widget within the SPARK-FW, the SPARKFactory would have to be modified in two places, once to add the new widget type to the list, and once to add the widget's constructor to the method that creates the widget.

```
1 - // class declaration
2 - function Pane (in_node) {
3 - }
4 - Pane.prototype = new Container;
5 - Pane.prototype.CGUI_TYPE = "PANE";

6 - // self registration
7 - SPARK.registerWidgetType( Pane.REGEX, Pane );
```

Figure 5.10: Widget Self Registration
The above shows an example of a SPARK compliant widget registering itself with the SPARK-FW. The registerWidgetType method will be called when the source file is included in the SVG document.

An alternative to modifying the SPARKFactory when new widgets are added is to register the widget type and constructor with the SPARK-FW when they are included in the SVG document (Figure 5.10). This approach has two main advantages. First, there is no need to modify the SPARKFactory to add a newly created widget. This means that the SPARKFactory can be treated as a static framework class that need not be modified by developers, which makes it easier to add a new widget to the Framework. Second, only the widgets that are used in the application are registered with the Framework, which saves including unused widgets or having to comment out references to them in the SPARKFactory. While this is an implementation detail that doesn't fundamentally alter anything about the SPARK-FW, this approach seems to work quite well and has been used in all the examples produced as part of this project.

There is no explicit imperative mechanism for creating widgets within the SPARK-FW, however they could be created by calling the constructor directly or calling the createWidget method of the SPARKFactory.

5.4.3 Adding functionality to a SPARK widget

The SPARK-FW uses the SPARKDecorator to attach Commands to the widgets at the time they are created by the SPARKFactory. The command pattern [21] encapsulates an action in an object whose functionality can be executed. The motivation for using commands in the SPARK-FW is

that this makes it possible for application designers to easily add logic to an otherwise generic framework.

On startup, after the widgets have been created by the SPARKHelperFactory, the SPARKDecorator is responsible for adding the appropriate command(s) to the newly created widgets, based on the widget's id. This means that a new decorator must be written for each application. An alternative to writing a new decorator for each application is to allow the widget to be decorated declaratively at the time that the UI is designed.

```
1 - <g id="b2" transform="translate(300,100)"
2 -     class="SPARK atom Button Style1">

3 -     <desc>Button 2</desc>
4 -     <metadata>bar</metadata>

5 -     <SPARKExt:decoration>Message</SPARKExt:decoration>

6  -    <g><!-- define view -->
7  -        <elipse rx="50" ry="25">
8  -            <animate attributeType="xml" attributeName="ry"
9  -                from="25"' to="35" dur="0.2s"
10  -               begin="b2.mouseover" fill="freeze"/>
11 -            <animate attributeType="xml" attributeName="ry"
12 -                from="35" to="25" dur="0.2s"
13 -                begin="b2.mouseout" fill="freeze"/>
14 -        </elipse>
15 -    </g>
16 - </g>
```

Figure 5.11: Declarative Decoration
The above code fragment shows the use of the "SPARKExt:decoration" element (line 5). When this Button is clicked, the method "Message" will be called. The "decoration" element is created in the "SPARKExt" namespace because it is not a valid SVG element.

This declarative decoration was enabled for trial purposes as part of a SPARK Extension by placing a decoration tag inside the widget declaration (Figure 5.11). The decoration tag specifies the name of the command with which to decorate the widget and provides any arguments to the command constructor as attributes. This extension required changing the SPARKFactory and SPARKDecorator to have them look for and process the decoration tag.

5.4.4 More concise widget declaration

The example button declaration presented in Figure 6 is quite verbose. Now imagine using that notation to describe an entire application. How long would the resulting SVG document be? Two different techniques were explored in an effort to make the GUI declaration more concise. Ultimately, the correct way to do this will be through the use of sXBL elements, but until the

specification is fully defined and there is a stable SVG viewer which supports sXBL other approaches are needed.

```
1 - <g id="b2" transform="translate(300,100)"
2 -     class="SPARK atom Button Style1">

3 -     <desc>Button 2</desc>
4 -     <metadata>bar</metadata>

6 -     <CScheme:body label="Lazy Button"/>

7 - </g>
```

Figure 5.12: External View
In the above example the view (Lines 5-16) of the original "Button" definition in Figure 5.6 has been replaced with a placeholder in a separate namespace. On startup the document will be parsed by a tag replacement method, and where the body tag is encountered it will be replaced by the appropriate view as determined by the class of the containing parent element (in this case "SPARK atom Button"). Parameters related to the construction of the view are passed in as attributes on the body tag. In this case the label for the button has been provided as a parameter.

The first technique involved moving the view to a separate file and replacing it with a placeholder in the SVG document. This placeholder is replaced by the externally defined view at the time that the SVG Document is loaded (Figure 5.12). This technique required the creation of an additional helper class to parse the document when it loads and replace all placeholders with the correct view. This technique has the advantage of being relatively simple and flexible as any necessary parameters can be set as attributes on the placeholder. However it requires the addition of a new custom tag and another helper class.

The second mechanism by which the view could be extracted is through use of the "use" tag. The "use" tag allows an SVG fragment to be defined and then reused throughout the document. This allows the view to be extracted without the need for a new tag and helper class, however it has a fairly serious limitation. While the referenced SVG fragment is rendered as if it was at the specified location in the DOM, the inserted fragment is actually part of a separate DOM and not accessible for manipulation via scripting. This makes it impossible to do things such as adjust the properties of the view to reflect state changes in the widget (though the widget can be styled via CSS) or to initialize the widget on creation. So this technique is best suited to creating templates that will never be changed.

A workaround for this limitation is to replace the use element in the DOM with the actual content that it references; however the benefits of this approach rather than the external definition are not clear (this approach was employed later in the plastic clock experiment) as the end result is effectively the same. In fact, this approach might be the worse of the two as it may not be desirable to replace all "use"' elements in the document and so some mechanism for distinguishing between

"use"' elements would be required.

5.4.5 Interoperability

At this point the CGUI button can be said to be SPARK compliant. It extends from the SPARK framework, implements the CommandHolder interface and uses Commands to perform its task, implements the Observable pattern to allow other widgets to receive notification when it receives user input and can be created declaratively by the SPARK factory. Now, the question becomes, has it achieved interoperability with other SPARK widgets? After all, interoperability was the major motivation for the SPARK-FW.

The new S-CGUI widget was used within the existing SPARK examples simply by substituting the old SPARK Button source for the new S-CGUI Button source and including the necessary CGUI sources. The only difficulty was an ECMAScript error that appeared when the SPARK Window containing the Button's was minimized. This turned out to be because the SPARK Buttons had undocumented show and hide methods rather than due to any fault in the S-CGUI Button.

5.5 Guidelines

There were four guidelines identified in the answers to the above questions that can be employed when making an existing widget set SPARK compliant. These include:

5.5.1 Classifying the existing widgets

The SPARK hierarchy provides two base widgets that should be extended when creating a new widget. In order to make a widget set SPARK compliant the existing widgets must be classified as either Atoms or Containers. This should be a straight-forward process as an Atom will not contain other widgets and may receive user input, while a Container may contain other widgets and is unlikely to accept user input. Another way to think about this is that the atom will generally provide the target for user input.

5.5.2 Inheriting the SPARK functionality

After classifying the existing widgets it is necessary to modify them so that they extend the appropriate SPARK class, either atom or container. If the existing widgets don't inherit from an existing hierarchy, this can be accomplished by setting the widgets prototype to be that of the extended class, and calling the super classes' constructor. If there is an existing hierarchy, then the functionality inherited from the existing hierarchy can be delegated to an another class (Figure 8). The "replace inheritance with delegation" refactoring is useful for this exercise.

5.5.3 Modifying the widget constructor

Having made the SPARK functionality available to the newly created widget and tested that it is accessible imperatively, it is time to alter the widgets constructor to allow it to be created declaratively within the SPARK framework. This means changing the widget's constructor so that it takes the root of the DOM sub-tree describing the widget as an argument and then binds to the tree as needed to provide necessary interaction.

It is desirable to keep something close to the old imperative syntax when modifying the constructor to ease migration of applications based on the old widget set. One way to do this is by separating construction of the view (i.e. DOM sub-tree) from the construction of the widget if it is not already separate. In doing so, ensure that any parameters which were previously passed to the widget constructor are passed into the method which returns the view, and add an init method to the widget that can be called with the remaining arguments to initialize the widget. Now, when the widget is created imperatively the view can be constructed and passed to the widget constructor as an argument, and when the widget is created declaratively the view and any initialization parameters can be extracted from the xml document describing the interface. The extract method refactoring is useful for moving view construction and initialization functionality out of the constructor.

5.5.4 Adopting use of the Command Pattern

The final step will be to modify the widget so that any code that should be executed in response to user input is executed via a Command. This can be accomplished by converting objects that implemented the Event Handler interface directly into Commands, though in the case of a complex Event Handler it might make sense to split this into a number of Commands. Then, these commands can be added to the widget using the addCommand method rather than registered as event listeners on the SVG Elements. If a method was used as an event handler previously rather than an object implementing the Event Handler Interface, that method can be moved to the execute method of the new Command, or split into several Commands just as the handleEvent method could be split.

5.6 Proposed Extensions

Three possible extensions to the SPARK framework were identified and explored as part of this project. The proposed extensions were presented as part of the paper at SVG Open 2005, but never formalized as part of the SPARK-FW specification. However, they proved functional and valuable in the two sample applications described briefly below (Section 5.7) in that they decreased the complexity of the SVG Document and required less code to be written to implement the

applications:

- An extension that allows new widgets to be registered with the SPARK framework when they are included in the SVG document, thereby eliminating the need to update the SPARK factory for each new widget type created (Fig 5.10).
- A declarative decoration mechanism for adding commands to the widgets in the XML definition of the UI rather than imperatively in the ECMAScript, which eliminated the need to write a new decorator class for each new application (Fig 5.11).
- An extension that allows the use of externally defined views or templates when declaring a widget, making the declaration more concise and eliminating much of the redundancy present in the original SPARK examples. Though this is only likely to be useful until sXBL is fully realized (Fig 5.12).

5.7 Sample Applications

There were two sample applications produced as part of this exercise[2]. The first of these, the "Train Game" is a simple grid based game that makes use of a number of S-CGUI Buttons with custom skin's designed to look like train cars. The second of these, the "Plastic Clock" is a simple grid based application that displays clocks for different time zones, and depending on a number of user definable parameters my display the clocks side by side or superimposed on top of each other. Each of these is contained in an S-CGUI Pane, another widget that was converted to test the above guidelines.

The "Train Game" was developed to experiment with the use of externally defined skins and to test the simple S-CGUI widgets in a real application (Figure 5.13). The object of the game is to return the scrambled train cars to their initial state and so that they can leave the yard. The train cars in the application have been implemented as S CGUI buttons with custom, externally defined skins. They were implemented as Buttons because it is intended that they will receive user input (a click event) and move in response to this input if there is a move available. The yard was implemented as a Pane as it has been used to contain and layout the Buttons.

Each car in the train yard has a Command associated with it that causes it to move in response to the click event. A better implementation might have been to associate the Command with the yard itself, and then have the click on the car propagate up to the yard. This would have involve the creation of less Command objects, and provided the yard with more responsibility for controlling the layout of the cars. Currently the behavior of the Buttons in the yard was handled by

[2]The functioning examples are available online, however, they require internet explorer with the Adobe SVG Viewer. `http://homepage.usask.ca/~ctl271/cgui/examples/cgui_spark.shtml`

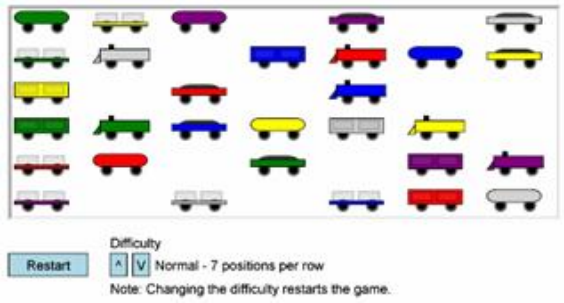

Figure 5.13: Train Game
Each car in the train game is an S-CGUI button with the appropriate skin applied to it. When the Buttons are clicked, they move behind the car that should precede it in the train, or move to the front of the line in the case of an engine if the space is available. Once all the cars have been arranged, either properly or there are no more valid moves, the game ends and a little animation moves the complete trains out of the yard.

an associated class called by the Buttons rather than through the Pane. The possibility of having the event propagate up to the yard and executing the Command there should be explored further in the next round of development on the Train Game.

The second application of the SPARK compliant CGUI widgets was to create a "Plastic Clock" (Figure 5.14). Plasticity is the measure of how well a user interface adapts itself to different displays, and the clock was a little experiment to show plasticity in SVG based applications. An SVG UI can be said to be trivially plastic based on the ability to infinitely scale an SVG graphic, however this experiment sought to show how the interactive nature of SVG allows more than a trivially plastic UI.

The original plastic clock (FlexClock) [Grolaux2001] was written as a replacement for the default XClock in the Unix operating system. It was designed to show the time in a number of formats depending on the size of the window it was displayed in. For instance, it might show a digital clock at one size, an analog clock at another size, and an analog clock with a one month calendar at a third size. The SVG plastic clock takes a slightly different approach. This clock was designed to display multiple time zones and only in an analog display. The clock takes advantage of transparency in the SVG specification to overlay clocks showing different time zones on top of each other when there is not enough space to display them beside one another. The clocks will always expand to take the maximum available space and have a user specifiable minimum size which controls when the clocks scale to fit the current cell rather than overlaying each other.

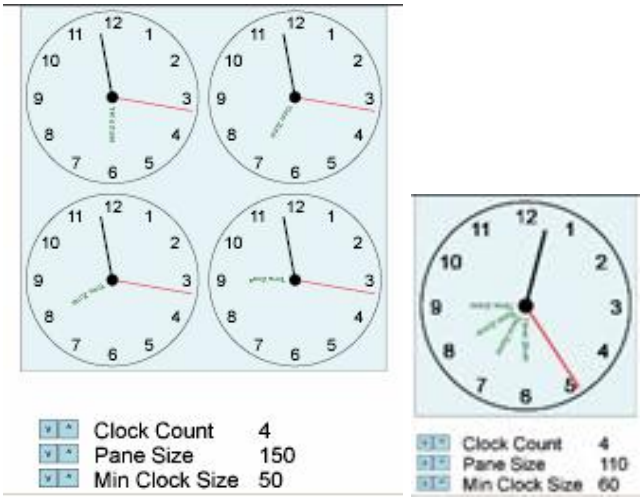

Figure 5.14: Plastic Clock
The left screen shot shows the plastic clock configured to display four clocks, each with a different timezone. As the Pane is resized the layout of the clocks changes based on the configurable number of clocks and minimum clock size. For instance, as the Pane becomes smaller, the clocks will morph into one larger clock that occupies the whole pane (right).

5.8 Conclusions

This work developed a simple and concise set of guidelines that should facilitate the quick and easy conversion of an existing SVG widget to a SPARK compliant widget. Such an effort has been shown to result in widgets which are interoperable with other SPARK compliant widgets as demonstrated by the use of the SPARK compliant CGUI widgets with the original SPARK sample widgets.

The three proposed extensions to the SPARK framework have been tested in the working examples produced for this paper and seem likely to make valuable additions to the framework once they've been suitably tested and refined.

The conceptual ease with which existing widgets can be converted to SPARK compliance and then used in a real application bodes well for the general applicability of the SPARK-FW for web application development. One real hindrance to the easy conversion of an existing widget set is the lack of a good ECMAScript editor with tool support for the refactoring the source.

5.9 Future Work

There are a number of future work items that require little explanation or have already been explained in the proceeding text, for instance:

- The proposed extensions need to be further developed to ensure that they are robust enough for general use.

- The guidelines for converting an existing widget set will be further developed and expanded as additional widget implementations are converted.
- As the above tasks progress it is certain that more useful extensions to the SPARK framework will be identified.

There are also a number of more interesting items that arise as a result of this effort. These are briefly discussed in the concluding text.

5.9.1 Server-side generation of widgets

The toy applications presented here were both fun and interesting, but they don't really capitalize on the benefits offered by SPARK compliance. For instance, the ability to create widgets declaratively means that it will now be possible to create new views on the server-side. This ability will make it simpler to track the state of the widgets in a database, which this can be used together with some sort of session tracking to save the state of an application so that the user can return to the application in the state from which they last accessed it. How best to track the state of SVG-based web-applications has been a long standing question in the SVG community.

This implies the need to add an API for generating the widgets on the server-side to the SPARK framework, and a messaging framework for tracking the widget state on the server, thereby adding persistence to the widgets. In an environment which didn't allow client side scripting to modify the view, for instance SVGT, this messaging framework could be used to redraw the application and return a new view each time a change was requested, much like current web applications generate a new bitmap and return it to the client to allow interactive visualizations.

5.9.2 Groupware

An interesting addition to the SVG 1.2 specification is sockets. This makes more interactive client server applications possible because the server can now easily push data to the client rather than waiting for the client to poll for new information. Socket based communication together with the messaging framework suggested above could be used to allow distributed, collaborative tools based on SVG to communicate. This would see the messages used to track the state of a widget on the server used to communicate change of state messages to a collaborator's computer, thereby enabling SVG based groupware.

5.9.3 Integration with other technologies

It has often been suggested that SVG could be used to render the widgets defined as part of the XForms specification. Now that the SPARK framework makes it relatively simple to create interoperable widgets, whether or not XForms widgets could be easily created within the SPARK

framework needs to be evaluated. One method for doing this might be to use the XForm widget definition as the model for the resulting widget and the existing framework to provide a view for it.

This is an important step as one of the advantages of SVG over similar proprietary formats is its ability to interact with other standards in a mixed namespace document. Up till now, due to a lack of compliant browsers, SVG developers have been force to work primarily within a plugin environment, which has restricted their ability to mix XML technologies. However, with the recent announcement that Opera supports SVGT and that Mozilla 1.1 will support the full SVG specification natively, a whole new world of opportunities become available. Evaluating the degree to which SPARK can take advantage of the Xforms specification seems like a manageable first step.

5.9.4 Use new standard functionality

sXBL is an SVG specific subset of the XML Binding Language that is important from the SPARK perspective as it allows the easy creation of interactive code templates. Unlike the "use" element, the DOM of a template added via the sXBL framework is accessible for manipulation. This should make it easier to template widgets and to modify their appearance without the "replace use hack" described earlier. The question is, how easy to modify SPARK compliant widgets to take advantage of the opportunities presented by sXBL?

5.9.5 AJAX

Ajax is an emerging approach to web application development that integrates a number of existing technologies XHTML+CSS, DOM, XML+XSLT, XMLHttpRequest and ECMAScript to provide more fluid and flexible web interfaces. It does this by eliminating the page reloads that existed as part of the old web paradigm. It will be interesting to see how the experiences of the Ajax community influence SVG based web application design, and how SVG might become part of this model. It is particularly interesting to the author because on the surface the approach taken by the Ajax community is almost exactly what was done in the Brassica / Arabidopsis Comparative Genome Browser when it was developed. With the possibility of moving SVG applications out of the plugin environment in the not too distant future, the Ajax paradigm could play a role the development of future SVG based applications.

5.9.6 Plasticity

An SVG based user interface should automatically do well with regard to plasticity as the UI can be scaled for free. Thus it should be possible to design an interface for an arbitrary display resolution and then scale the interface using the viewBox of the SVG document. However, for extreme changes

in resolution, for instance from a desktop display to a large display or cellular phone display, the UI is unlikely to be usable even if it is scaled to reflect the resolution of the display. The question to be answered here is how, if at all, SVG can be used to create standard, plastic widgets that will function on a variety of display formats.

One possible solution is the use of XSLT to adjust the interface on the server-side. This transformation could happen before the UI is sent to the client in a web environment, or to simply generate the appropriate depending on the display resolution when the application is started in a standalone environment. However, even more interesting would be widgets that were smart enough that if the display resolution changed the UI would be transformed.

Chapter 6

Summary and Conclusions

6.1 Summary

This thesis described the development of BioViz, a novel comparative genome browser developed for the *Brassica-Arabidopsis* crop-model pair. The description was separated into three parts roughly coinciding with existing publications related to functionality, implementation, and impact of the browser. Given that the existing papers focused more on the use of SVG than the bioinformatics, the section related to functionality was updated to include a description of the data representations used in the browser and a background chapter was added to provide relevant bioinformatics information.

6.1.1 Functionality and data representation

BioViz allowed the user to view short *Brassica* sequences, including expressed sequence tags (ESTs), activation tagged sequences, serial analysis of gene expression tags (SAGE), and oligonucleotide sequences in the context of the *Arabidopsis* genome. By aligning these short sequences with the *Brassica* genome it provided context for the sequences, without which they are relatively meaningless. In the case of the EST sequences, SAGE tags and oligo sequences, positioning them relative to the *Arabidopsis* genome provided access to the Arabidopsis gene annotations. Further, aligning the ESTs with the genome sequence provided a provisional clustering of related sequences. Aligning the SAGE tags with the genome sequence allowed users to look for clusters of up and down regulated genes - so called "gene islands". Aligning the oligo sequences with the genome allowed the user to see which portions of the oligo sequences were similar with *Arabidopsis* and which were *Brassica* specific.

This required the creation of representations for each of the short sequences and also of the *Arabidopsis* genomic information. The *Arabidopsis* chromosomes were represented as horizontal rectangular bars, divided into groups corresponding to the bacterial artificial chromosome (BAC) sequences from which the full chromosome sequences were sequenced. In BioViz, the BACs served to split the chromosomes into regions within which the user could drill down. At the BAC-level, the user was presented with a representation of the genes overlaid on top of the rectangle representing the BAC. The regions of overlap with the neighbouring BACs are also presented, which allows users

to identify genes that are in the overlap region. At this level, the user is able to navigate left or right along the chromosome to the neighbouring BACs, to view the sequences with homology along this region of the genome, or to drill-down further and access information related to the represented genes or *Brassica* sequences.

It is at the BAC level where the strengths of SVG for this application become apparent. SVG is a vector graphics format, and given this fact, users can zoom in on the BAC to whatever level of detail they desire - the images are drawn to correspond to one base pair per pixel, and users can zoom into the point where they can see the individual base pairs. This allows them to see even single basepair differences when comparing a stack of ESTs, differences that might be exploitable for further analysis or applications. That the image is transformed on the client-side means that they can make as many fine-grained changes to the view as desired in real-time, whereas making changes with the web-based browsers is more difficult because they typically only allow gross position or scale changes.

The arrangement of features relative to the center-line in BioViz is distinct from the track-based arrangement used in the standard web-based genome browsers. We argue that the arrangement taken in BioViz makes it easier to tell when sequences are on different strands, especially in high-level views. However, the addition of an arrow to the features in BioViz would make the orientation of a feature explicit in BioViz. The arrangement used in BioViz is also less amenable to the display of multiple tracks simultaneously as is done in the standard web-based genome browsers. The split display used in BioViz would result in an ever increasing distance between the + and − orientation tracks where multiple tracks displayed. However, for the purposes of comparing sets of sequences with the genome one at a time, we feel there are advantages to the split display.

In some sense the BAC View is artificial in that the BACs are an artifacts of the sequencing process. One could imagine replacing BACs with length based segments in a future version of the browser or if the browser were adapted for use with another organism. For performance reasons, it is likely undesirable to simply use the whole chromosome for the detailed view, however, this would need to be explored future.

6.1.2 Implementation

BioViz was implemented as a client-server application wherein the client presented the view of the data returned from the server in response to client requests. BioViz uses a standard message format for client requests and server responses. The client-side of BioViz is implemented in ECMAScript and uses SVG to provide the user interface elements and data representations. A custom graphical user interface (GUI) library was developed to provide the user interface elements found in BioViz. The server-side message handling framework is implemented in Perl and it allows plugins to be registered with the framework for various client requests. The plugins implemented for BioViz

utilize the BioPerl library to handle file IO and parse output from standard bioinformatics applications such as BLAST. The Perl SVG module is used to create the SVGDocument fragment that is returned to the client containing content.

The AJAX-like implementation employed in BioViz was significantly ahead of its time, and is only now (2008) being implemented in other mainstream web-based genome browsers. The use of SVG facilitates this style of development because not only is it possible to return a new image without a page refresh (as in an AJAX-style bitmap-based genome browser), but it is also possible to return only a portion of a document and dynamically update the existing document. This further reduces the impact of a refresh on the user and gives an even smoother user experience. The ability to transform the content on the client-side is another advantage offered by SVG in that it avoids delays while waiting for updated images from the server, and should save bandwidth and reduce server load. The data reported in section 3.4.2 suggests potential significant savings in terms of required bandwidth and browser responsiveness from the use of client-side transformations together with a vector graphics format representation. Based on the stated motivation behind the implementation of "slave rendered support" in the GBrowse rewrite (2.0), we conclude that there is an appreciable cost to constantly rendering new views on the server-side and that reduced server load could be an additional benefit of client-side rendering and transformations.

6.1.3 Impact

BioViz has been in use at Agriculture and Agri-food Canada since its initial deployment circa 2002. Since then it has proven a valuable tool to provide researchers and technicians with access to the *Brassica* resources developed at AAFC. In that it was one of the first genome browsers available, and the first (and for several years only) browser targeted at *Brassica* / *Arabidopsis* we go so far as to claim it was an invaluable resource for the *Brassica* community.

The GUI library developed for BioViz has been used in a number of third party applications and motivated two different open source projects to continue the work started for BioViz. The SVG Programmers Application Resource Kit (SPARK) project was started to develop standards for SVG-based web applications. To that end the SPARK GUI Framework was developed and the relationship between SPARK and CGUI was explored in a paper presented at SVG Open 2005. The CSVGUI project was started from the main CGUI project and with library upkeep being performed by one of the main CGUI contributors as of 2007.

6.2 Future Work

GBrowse appears to have become the *de facto*-standard genome browser. When GBrowse 2.0 is completed, it will need to be evaluated to see whether or not the usability concerns we sought to

address in BioViz are still valid. If so, then it is likely that BioViz should be retired. Our findings in section 3.4.2 suggest that the user experience in GBrowse should be substantially improved in the proposed AJAX-based implementation in that it will by reduce or eliminate the users need to reorient after each page refresh. The addition of the slave rendering support might allow the addition of sufficient support servers to make the time required to re-render views insignificant, thought hardware and bandwidth costs might still be factors when considering offloading some of the computational effort to the client.

If the usability issues are not addressed, or for users who appreciate the "comparative" focus in BioViz as compared to GBrowse, then it might be worthwhile to update the BioViz interface to work with the new browsers, as well as writing database adapters to allow BioViz to run off an existing GBrowse installation. Such adapters were started, but not completed as part of the BioViz server-side rewrite. Were they completed BioViz could be used as a web interface for any database currently available in GBrowse, making BioViz immediately and generally applicable to organisms other than *Brassica* and *Arabidopsis*.

The version of the CGUI library available from the CSVGUI project has been updated, so in theory making BioViz work with the native SVG support available in modern browsers is simply a matter of updating the library. However, it is probable that there have been API changes since the library used in BioViz was last updated that will need to be resolved. When last the author explored native SVG support, a number of implementation differences between browsers were identified that might further complicate the task.

"Accessibility" is a hot topic in web-development, it refers to the practice of making websites usable by people of all abilities and disabilities. It is especially relevant in government because as a result of Treasury Board of Canada Standards for accessibility, applications such as BioViz cannot be placed online. Eventually, if BioViz is not updated, it is likely that we will be forced to take it down. An interesting future project would be to explore the requirements for developing an "accessible" comparative genome browser. Given that SVG graphics are XML based, there are unique possibilities for rendering them as text that could be read by a screen reader for the visually impaired. Making images "accessible" is one of the major hurdles that has been encountered by developers.

6.3 Hindsight

Looking back on the project at completion, there are a few things that should have been implemented differently in BioViz. The server-side rewrite addressed a large portion of the problem through the introduction of a config file, however, the client-side also needed to be reworked to capitalize on the changes. When BioViz was first implemented, the menu options (e.g. data sets

available for loading / searching) were hardcoded in the JavaScript. This meant that adding a new dataset required the JavaScript to be updated each time. Ideally, the menu options should have been provided by the server, and then after the rewrite they could have automatically reflected the data sets available in the config file.

The multi-windowed format employed in BioViz may or may not have had value. We opted for that style because the "desktop" style interface is a familiar one, it provided a more "seamless" interface by avoiding the use of popups, and it allowed us to avoid limitations on inter-browser communication imposed by the "sandbox" in which plugins execute. However, the "familiarity" argument probably doesn't apply given that most web application interfaces are quite different from that of BioViz, and users seem quite content with the use of popups in web applications so those are no longer an issue (if they ever were). Finally, given native support for SVG, any arguments related to plugin limitations are moot.

That being said, the author would certainly maintain the AJAX-style interface and client-side transformations used in BioViz. These appear to have been two major strengths of BioViz which were enabled by the use of SVG. SVG was and still is an exciting technical specification with the potential to enable a radically different web experience. Now that native browser support for SVG is maturing, the possibilities become even greater as it becomes possible to create mixed namespace documents utilizing SVG, HTML and other emerging specifications all together in a single page. If BioViz were to be re-implemented at this point in time by the author, SVG would still be used for the data representations, however, the user interface would likely be a mix of HTML and SVG.

On the server-side, rather than creating a custom message handling framework, the author would explore open source web applications frameworks. The author might chose to use Java rather than Perl at this point in time. However, a deciding factor would be the degree to which BioJava provides support for GFF format databases (the format used in GBrowse). The author feels strongly that providing the ability to work with existing GBrowse installations would be a huge boost to the popularity of BioViz. It would also ensure that users could use existing tools which are being developed around this standard format.

References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, Oct 1990.

[2] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res*, 25(17):3389–3402, Sep 1997.

[3] V. S. Bhinu, U. A. Schafer, R. Li, J. Huang, and A. Hannoufa. Targeted modulation of sinapine biosynthesis pathway for seed quality improvement in Brassica napus. *Transgenic Res*, July 2008.

[4] J. A. Blake and M. A. Harris. The Gene Ontology (GO) project: structured vocabularies for molecular biology and their application to genome and expression analysis. *Curr Protoc Bioinformatics*, Chapter 7:Unit 7.2, Nov 2002.

[5] P. Bonizzoni and G. D. Vedova. The complexity of multiple sequence alignment with SP-score that is a metric. *Theor. Comput. Sci.*, 259(1-2):63–79, 2001.

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation 1.0, World Wide Web Consortium (W3C), Nov 2008.

[7] D. Bulterman, G. Grassel, J. Jansen, A. Koivisto, N. Layaïda, T. Michel, S. Mullender, and D. Zucker. Synchronized Multimedia Integration Language (SMIL). W3C Recommendation 2.1, World Wide Web Consortium (W3C), Dec 2005.

[8] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM J. Appl. Math.*, 48(5):1073–1082, 1988.

[9] J. Clark. XSL Transformations (XSLT). W3C Recommendation 1.0, World Wide Web Consortium (W3C), Nov 1999.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT Press, Cambridge, MA, USA, 2001.

[11] O. Couronne, A. Poliakov, N. Bray, T. Ishkhanov, D. Ryaboy, E. Rubin, L. Pachter, and I. Dubchak. Strategies and tools for whole-genome alignments. *Genome Res*, 13(1):73–80, Jan 2003.

[12] S. DeRose, E. Maler, and D. Orchard. XML Linking Language (XLink). W3C Recommendation 1.0, World Wide Web Consortium (W3C), June 2001.

[13] ECMA. ECMAScript Language Specification. Standard 262, ECMA, Dec 1999.

[14] R. Edgar. Muscle: a multiple sequence alignment method with reduced time and space complexity. *BMC Bioinformatics*, 5(1):113, 2004.

[15] D. F. Feng and R. F. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *J Mol Evol*, 25(4):351–360, 1987.

[16] J. Ferraiolo, F. Jun, and D. Jackson. Scalable Vector Graphics (SVG) 1.1 Specification. W3C Recommendation, World Wide Web Consortium (W3C), Jan 2003.

[17] A. Fettes and P. Mansfield. SVG-based user interface framework. Tokyo, Japan, 2004. SVG Open 2004.

[18] P. Flicek, B. L. Aken, K. Beal, B. Ballester, M. Caccamo, Y. Chen, L. Clarke, G. Coates, F. Cunningham, T. Cutts, T. Down, S. C. Dyer, T. Eyre, S. Fitzgerald, J. Fernandez-Banet, S. Graf, S. Haider, M. Hammond, R. Holland, K. L. Howe, K. Howe, N. Johnson, A. Jenkinson, A. Kahari, D. Keefe, F. Kokocinski, E. Kulesha, D. Lawson, I. Longden, K. Megy, P. Meidl, B. Overduin, A. Parker, B. Pritchard, A. Prlic, S. Rice, D. Rios, M. Schuster, I. Sealy, G. Slater, D. Smedley, G. Spudich, S. Trevanion, A. J. Vilella, J. Vogel, S. White, M. Wood, E. Birney, T. Cox, V. Curwen, R. Durbin, X. M. Fernandez-Suarez, J. Herrero, T. J. P. Hubbard, A. Kasprzyk, G. Proctor, J. Smith, A. Ureta-Vidal, and S. Searle. Ensembl 2008. *Nucleic Acids Res*, 36(Database issue):D707–14, Jan 2008.

[19] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[20] K. A. Frazer, L. Pachter, A. Poliakov, E. M. Rubin, and I. Dubchak. Vista: computational tools for comparative genomics. *Nucleic Acids Res*, 32(Web Server issue):W273–9, Jul 2004.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[22] GMOD. GBrowse 2.0 HOWTO. `http://gmod.org/wiki/GBrowse_2.0_HOWTO#Introduction`.

[23] R. C. G. Holland, T. A. Down, M. Pocock, A. Prlic, D. Huen, K. James, S. Foisy, A. Drager, A. Yates, M. Heuer, and M. J. Schreiber. BioJava: an open-source framework for bioinformatics. *Bioinformatics*, 24(18):2096–2097, Sep 2008.

[24] T. Hubbard. Biological information: making it accessible and integrated (and trying to make sense of it). *Bioinformatics*, 18 Suppl 2:S140, 2002.

[25] W. Just. Computational complexity of multiple sequence alignment with SP-score. *Journal of Computational Biology*, 8(6):615–623, 2001.

[26] W. J. Kent. BLAT–the BLAST-like alignment tool. *Genome Res*, 12(4):656–664, Apr 2002.

[27] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The human genome browser at UCSC. *Genome Res*, 12(6):996–1006, Jun 2002.

[28] S. Kumar. Developing a GIS-based geo-portal with scalable vector graphics (SVG) for accessing environmental information of baden-württemberg. Master's thesis, University of Applied Sciences in Stuttgart, March 2003.

[29] C. T. Lewis. Brassica Arabadopsis Genomics Initiative (BAGI). `http://brassica.agr.gc.ca`.

[30] C. T. Lewis. CGUI source page. `http://homepage.usask.ca/~ctl271/cgui/index.shtml`.

[31] C. T. Lewis. Experiences creating a SPARK compliant widget set. Enschede, Netherlands, August 2005. SVG Open.

[32] C. T. Lewis, C. L. Abbott, J. T. Chapados, R. D. Peters, H. W. Platt, M. D. Coffey, and C. A. Lévesque. Development of a snp based marker system based on variable microsatellite flanking regions for phytophthora infestans. In *2007 Annual Meeting of CPS-SCP (with Plant Canada 2007)*, page 139, Saskatoon, SK, Canada, 2007.

[33] C. T. Lewis, A. Fettes, and C. B. Peto. SVG programmers' application resource kit (SPARK). `http://spark.sourceforge.net/`.

[34] C. T. Lewis, S. Karcz, A. Sharpe, and I. A. P. Parkin. Bioviz: Genome viewer. Zurich, Switzerland, July 2002. SVG Open.

[35] C. T. Lewis, P. Mansfield, A. Fettes, and G. MacDonald. SPARK - SVG programmers application resource kit (presentation). Vancouver, BC, Canada, July 2003. SVG Open.

[36] C. T. Lewis, A. Sharpe, D. Lydiate, and I. A. P. Parkin. The Brassica/Arabidopsis comparative genome browser: A novel approach to genome browsing. *Journal of Plant Biotechnology*, 5(4):197–200, Dec 2003.

[37] K. Lindsey. Kevlindev. `http://www.kevlindev.com/`.

[38] C. G. Love, J. Batley, G. Lim, A. J. Robinson, D. Savage, D. Singh, G. C. Spangenberg, and D. Edwards. New computational tools for Brassica genome research. *Comp Funct Genomics*, 5(3):276–280, 2004.

[39] H. Mangalam. The Bio* toolkits–a brief overview. *Brief Bioinform*, 3(3):296–302, Sep 2002.

[40] U. N. Genome analysis in Brassica with special reference to the experimental formation of B. napus and peculiar mode of fertilization. *Jpn J Bot*, 7:389–452, 1935.

[41] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[42] C. Notredame. Recent evolutions of multiple sequence alignment algorithms. *PLoS Comput Biol*, 3(8):e123, Aug 2007.

[43] C. Notredame and D. G. Higgins. SAGA: sequence alignment by genetic algorithm. *Nucl. Acids Res.*, 24(8):1515–1524, 1996.

[44] I. A. P. Parkin, A. G. Sharpe, D. J. Keith, and D. J. Lydiate. Identification of the A and C genomes of amphidiploid Brassica napus (oilseed rape). *Genome*, 38(6):1122–1131, 1995.

[45] E. Pennisi. Sequence: Plants join the genome sequencing bandwagon. *Science*, 290(5499):2054–2055, Dec 2000.

[46] C. B. Peto. GEMOS building management and security system's SVG interface. Zurich, Switzerland, August 2005. SVG Open.

[47] T. Pixley. Document Object Model (DOM) Level 2 Events Specification. W3C Recommendation, World Wide Web Consortium (W3C), Nov 2000.

[48] X. Qiu, S. Pallickara, and A. Uyar. Making SVG a web service in a message-based MVC architecture. Tokyo, Japan, September 2004. SVG Open.

[49] S. J. Robinson, J. D. Guenther, C. T. Lewis, M. G. Links, and I. A. P. Parkin. Reaping the benefits of SAGE. *Methods Mol Biol*, 406:365–386, 2007.

[50] D. Schepers. Web applications and compound documents leveraging SVG. Workshop on Web Applications and Compound Documents, San Jose, California, USA, June 2004. World Wide Web Consortium (W3C).

[51] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[52] P. T. Spellman, M. Miller, J. Stewart, C. Troup, U. Sarkans, S. Chervitz, D. Bernhart, G. Sherlock, C. Ball, M. Lepage, M. Swiatek, W. L. Marks, J. Goncalves, S. Markel, D. Iordan, M. Shojatalab, A. Pizarro, J. White, R. Hubley, E. Deutsch, M. Senger, B. J. Aronow, A. Robinson, D. Bassett, C. J. Stoeckert Jr, and A. Brazma. Design and implementation of microarray gene expression markup language (MAGE-ML). *Genome Biol*, 3(9):research0046.1–0046.9, Aug 2002.

[53] J. E. Stajich, D. Block, K. Boulez, S. E. Brenner, S. A. Chervitz, C. Dagdigian, G. Fuellen, J. G. R. Gilbert, I. Korf, H. Lapp, H. Lehvaslaiho, C. Matsalla, C. J. Mungall, B. I. Osborne, M. R. Pocock, P. Schattner, M. Senger, L. D. Stein, E. Stupka, M. D. Wilkinson, and E. Birney. The Bioperl Toolkit: Perl modules for the life sciences. *Genome Res*, 12(10):1611–1618, Oct 2002.

[54] L. D. Stein, C. Mungall, S. Shu, M. Caudy, M. Mangone, A. Day, E. Nickerson, J. E. Stajich, T. W. Harris, A. Arva, and S. Lewis. The Generic Genome Browser: a building block for a model organism system database. *Genome Res*, 12(10):1599–1610, Oct 2002.

[55] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucl. Acids Res.*, 22(22):4673–4680, 1994.

[56] TIGR. Evidence supporting annotated Arabidopsis gene models. `http://www.tigr.org/tigr-scripts/e2k1/arab_gene_phys_ev_classification.cgi`, 2004.

[57] A. L. Turinsky, A. C. Ah-Seng, P. M. K. Gordon, J. N. Stromer, M. L. Taschuk, E. W. Xu, and C. W. Sensen. Bioinformatics visualization and integration with open standards: the Bluejay genomic browser. *In Silico Biol*, 5(2):187–198, 2005.

[58] V. E. Velculescu, L. Zhang, B. Vogelstein, and K. W. Kinzler. Serial analysis of gene expression. *Science*, 270(5235):484–487, Oct 1995.

[59] I. M. Wallace, O. Orla, and D. G. Higgins. Evaluation of iterative alignment algorithms for multiple alignment. *Bioinformatics*, 21(8):1408–1414, 2005.

[60] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *J Comput Biol*, 1(4):337–348, 1994.

[61] J. R. Wortman, B. J. Haas, L. I. Hannick, R. K. J. Smith, R. Maiti, C. M. Ronning, A. P. Chan, C. Yu, M. Ayele, C. A. Whitelaw, O. R. White, and C. D. Town. Annotation of the Arabidopsis genome. *Plant Physiol*, 132(2):461–468, Jun 2003.